# BIBLIOTECA BASICA INFORMATICA

## PROGRAMAR EN C 29



INGELEK

# BIBLIOTECA BASICA INFORMATICA

## PROGRAMAR EN C 29

INGELEK

**Director editor:**
Antonio M. Ferrer Abelló.

**Director de producción:**
Vicente Robles.

**Coordinador y supervisión técnica:**
Enrique Monsalve.

**Redactor técnico:**
Alfredo B. García Pérez.

**Colaboradores:**
Casimiro Zaragoza.

**Diseño:**
Bravo/Lofish.

**Dibujos:**
José Ochoa.



ISBN del tomo: 84-85831-74-8
ISBN de la obra: 84-85831-31-4
Fotocomposición: Pérez Díaz, S. A.
Imprime: Héroes, S. A.
Depósito legal: M-16945-1986
Precio en Canarias, Ceuta y Melilla: 380 pts.

# INDICE

# PROLOGO

En un intento inicial de definirlo podríamos describir el lenguaje C como un lenguaje de programación de propósito general muy bien adaptado a la programación estructurada.

Al decir que se trata de un lenguaje de propósito general expresamos la idea de que no se trata de un lenguaje específico u orientado hacia algún tipo de aplicación en particular, como pueden ser los casos de los lenguajes COBOL (COmmon Business Oriented Language) y FORTRAN (FORmula TRANslation) orientados a la gestión comercial y al cálculo científico, respectivamente.

La idea de un lenguaje de programación de propósito general no se refiere sólo a la orientación del lenguaje hacia algún campo de aplicación, sino a la ausencia de instrucciones específicas para realizar determinadas tareas y, lo que es más importante, a la ausencia de las limitaciones (muchas veces artificiales) impuestas por los lenguajes de programación tradicionales.

Este hecho dota al lenguaje C de una potencia inusual: por una parte presenta un juego de instrucciones muy reducido, pero suficiente y fácil de aprender, y por otra ofrece muy pocas limitaciones en cuanto a su utilización.

C es un lenguaje utilizado tanto para la implementación de Sistemas Operativos y Lenguajes de alto nivel, como para la realización de Utilidades y Programas de Aplicación. No debemos olvidar que C es el lenguaje nativo del S. O. UNIX (aproximadamente un 95 por 100 está escrito en este lenguaje) ni que la mayoría de los lenguajes de programación soportados por UNIX (BASIC,

COBOL, FORTRAN y PASCAL) están íntegramente escritos en lenguaje C.

Es de destacar el elevado grado de portabilidad que ofrece el C, permitiendo que aplicaciones desarrolladas en él sobre un equipo con hardware de un determinado fabricante puedan funcionar sobre equipos de otros fabricantes con la realización de unas mínimas e incluso inexistentes modificaciones. Esto implica una virtual independencia del software con respecto del hardware y disminuye los costes de desarrollo software al ampliar el espectro de equipos sobre los que puede funcionar un determinado paquete de aplicación.

C soporta muy bien el empleo de programación estructurada, admitiendo el diseño de programas mediante la realización de bloques cada vez más complejos y refinados (diseño "de arriba-abajo" o top-down) e incorporando las estructuras de control básicas como WHILE, DO, UNTIL, SWITCH e IF-THEN-ELSE.

Por otra parte, el lenguaje de programación C posee características de relativamente bajo nivel, como son el manejo de direcciones de memoria, el tratamiento de campos de bit, el acceso a funciones a nivel de Entrada/Salida del sistema operativo (open, read, write, seek, close) y su implementación a base de librerías de subrutinas.

La mayoría de las funciones de C están realizadas en forma de subrutinas escritas en el propio lenguaje C, de modo que resulta natural su enriquecimiento mediante la inclusión de nuevas funciones, escritas a medida que son requeridas.

Para sentar ideas podríamos comparar en algunos aspectos al C con otros lenguajes de programación, como BASIC, PASCAL y ENSAMBLADOR, diciendo que C tiene casi la sencillez del BASIC, admite programación estructurada como el PASCAL y llega en ocasiones a un nivel tan cercano al hardware como el del ENSAMBLADOR.

Como contrapartidas, C presenta una sintaxis relativamente compleja y una dificultad de comprensión inicial, tal vez debida a la costumbre y adquisición de "vicios de programación" inherentes al uso de otros lenguajes.

En los primeros pasos al programar en lenguaje C el programador se siente un poco indefenso ante la aparente falta de potencia del lenguaje, que implica el uso de funciones de librería para casi todo aquello que está acostumbrado a hacer mediante instrucciones en otros lenguajes.

No deja de ser chocante para un programador enfrentarse al hecho de que para copiar un literal o cadena (string) sobre otro requiera una función de liberaría (strcpy), o que no existan instrucciones para el manejo de ficheros indexados (C sólo puede tratar ficheros de acceso secuencial y relativo), o incluso que no



existan funciones de formateado para Entrada/Salida de datos (como es el caso de las máscaras de impresión, por ejemplo "###.###,##").

Sin embargo, resulta trivial escribir una función que copie un literal sobre otro, es relativamente sencilla la escritura de una función de formateado con máscaras de impresión, están disponibles comercialmente unas excelentes librerías de manejo de ficheros indexados, bases de datos, formateadores de pantallas y todo tipo de utilidades de programación escritas íntegramente en lenguaje C, y que son totalmente transportables de un sistema a otro.

La conclusión que debemos extraer es que es mucho mayor la flexibilidad que se obtiene al manejar librerías de funciones que la que da el conjunto de instrucciones incorporadas por un lenguaje de alto nivel, aunque al precio de una mayor inversión inicial en desarrollo de software, hasta completar la escritura de las propias librerías de funciones. Esto es algo que no permiten todos los lenguajes de programación, o al menos no con la misma naturalidad que el lenguaje C.

Una de las limitaciones aparentes es su carencia de posibilidades para facilitar la multiprogramación, que podríamos definir como la posibilidad de partir un programa único en otros varios, ejecutándolos simultáneamente y dotándolos de un medio que posibilite la comunicación y cooperación entre ellos.

Sin embargo, el sistema operativo Unix suple esta carencia mediante las oportunas llamadas a funciones del núcleo del sistema. Allí donde no llega el lenguaje C por sí solo puede llegar ayudado por el núcleo de Unix, que podríamos ver como una especie de caja negra con la única misión de activar y desactivar procesos (un proceso no es más que un programa en ejecución), realizar operaciones de Entrada/Salida, permitir la intercomunicación de procesos y tratar adecuadamente las interrupciones y condiciones de excepción que se puedan presentar en el transcurso de la ejecución de un proceso.

Como el sistema operativo Unix está escrito en su mayor parte en lenguaje C, esto implica que las funciones primitivas del sistema (o system-calls, pequeñas funciones aisladas cuyo conjunto forma el sistema operativo) puedan ser utilizadas directamente por los programas en C que así lo requieran sin necesidad de realizar una interface con lenguaje ensamblador como viene sucediendo en la inmensa mayoría de los sistemas operativos.

Podemos afirmar que el sistema operativo Unix y el lenguaje de programación C son como dos partes de un todo indivisible: C es el lenguaje de programación idóneo para un entorno Unix, y Unix es el sistema operativo que más facilidades ofrece al empleo del lenguaje C. A través de la utilización conjunta de Unix y C es como se alcanza la máxima potencia y eficacia de ambos.

En los próximos capítulos presentaremos los orígenes del lenguaje de programación C y haremos una descripción detallada del lenguaje, sus tipos de datos, estructuras de control y funciones, con la inclusión de ejemplos en aquellos apartados donde favorezcan su comprensión. También comentaremos algunos aspectos interesantes del compilador de C y trazaremos una panorámica acerca del estado actual del lenguaje, sus tendencias, nociones de C avanzado, y una breve descripción de algunas librerías de funciones notables.

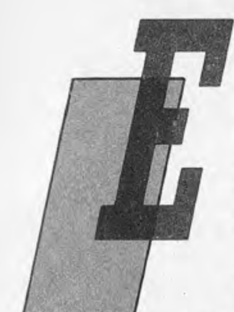
Todo esto con las naturales limitaciones de espacio y profundidad en la exposición impuestas por el carácter divulgativo de este texto. Para los lectores expertos o interesados en profundizar en el conocimiento del lenguaje C se ha incluido una amplia bibliografía.

Dada la escasez de libros sobre C en castellano (las pocas ediciones que existen son además traducciones) confiamos en que este libro les sirva de ayuda para dar sus primeros pasos en él y que seamos capaces de transmitir a nuestros lectores la inquietud y el deseo de profundizar en el conocimiento y dominio de un lenguaje tan lleno de posibilidades como es el C.



# CAPITULO I

## HISTORIA DEL C



El lenguaje de programación C fue desarrollado en la década de los setenta por Dennis Ritchie en los laboratorios Bell de la AT&T, en Murray Hill (New Jersey). Su historia corre paralela a la del desarrollo del sistema operativo Unix por Ken Thompson en los mismos laboratorios.

El nacimiento de Unix tuvo lugar por la necesidad de disponer de un sistema operativo con unas posibilidades similares a las de los grandes sistemas de la época (1970) y que pudiese funcionar en pequeños ordenadores de propósito general como los DEC PDP-11. Con esta finalidad, Ken Thompson desarrolló una primera versión de un minisistema-operativo al que denominó UNIX (frente a MULTICS, un s. o. de grandes ordenadores), escrito inicialmente en lenguaje ensamblador.

Posteriormente se reescribió esta primera versión en un nuevo lenguaje de programación denominado "lenguaje B", desarrollado por Ken Thompson e inspirado en el lenguaje BCPL, similar en algunos aspectos al Pascal de Niklaus Wirth.

Este lenguaje (B) fue posteriormente revisado por Dennis Ritchie, ampliándolo y obteniendo una nueva versión a la que denominó "lenguaje C", desarrollado a la vez que se reescribía UNIX en él.

Estaban lejos de imaginar, tanto Ken Thompson como Dennis Ritchie, lo que sucedería pocos años después con el sistema operativo Unix y el lenguaje C, convertidos en estándares de la industria de ordenadores.

Tras el desarrollo de las primeras versiones de Unix y C, se cedieron licencias de uso a distintas universidades norteamerica-

nas, que contribuyeron con sus mejoras y desarrollos posteriores a lograr una primera versión comercial del sistema operativo Unix hacia finales de los años setenta.

Habrían de pasar unos años hasta que Unix y C alcanzasen su posición actual, favorecidos por el tremendo desarrollo del hardware, capaz de conseguir microordenadores de 16 bits con la misma o mayor potencia que los antiguos miniordenadores de tan sólo una década atrás. Se hacía necesario un sistema operativo capaz de aprovechar la potencia de estas nuevas máquinas y la industria del software no estaba en condiciones de ofrecer ninguna alternativa válida por el momento.

Había que conseguir un sistema operativo de fácil implementación en un ordenador, potente en sus prestaciones, y lo suficientemente estándar como para ocupar el papel que el antiguo CP/M había jugado en el campo de los sistemas operativos para pequeños ordenadores personales.

Tras un tiempo de pruebas empezaron a salir al mercado las primeras máquinas trabajando en sistema operativo Unix, causando extrañeza el hecho de que en su mayor parte (cerca de un 95 por 100) estuviese escrito en un "extraño" lenguaje de alto nivel denominado "lenguaje C". El que no estuviese escrito en su totalidad en C se debía a que era necesario codificar en ensamblador las subrutinas de interface con el hardware y algunos bloques de código por razones de eficiencia en su utilización.

Una vez desarrollados Unix y C llegó la hora de escribir otros lenguajes de programación que pudiesen funcionar sobre Unix; fueron también desarrollados en C, lo mismo que bases de datos, programas de comunicaciones y todo tipo de utilidades. Sin darse cuenta, C se había convertido en una parte importantísima del sistema operativo Unix y prácticamente inseparable en su uso.

Llegados al momento actual observamos que la casi totalidad de los fabricantes de ordenadores han adoptado el sistema operativo Unix o derivados (como Xenix, Onyx, Zeus, Sinix, Venix, etc.) en sus equipos y que el lenguaje C se ha convertido en un denominador común en todos ellos.

Por si fuera poco, el C se ha mostrado como un lenguaje de una elevada portabilidad en su traspaso de programas entre ordenadores, lo cual es una razón para su empleo por parte de los fabricantes de software, al ampliar el horizonte de sus productos.

La tecnología actual ha desarrollado los primeros ordenadores de 32 bits capaces de funcionar con el sistema operativo Unix, siendo totalmente compatibles con sus hermanos "menores" de 16 bits, y ya se apuntan los primeros pasos hacia microordenadores de 64 bits, con posibilidades tan interesantes como el hecho de incluir procesamiento distribuido, siendo de esperar que también funcionen bajo el sistema operativo Unix.



Aparentemente estamos ante un estándar de mercado incuestionable, como es el sistema operativo Unix, llamado a ocupar entre los microordenadores dotados de posibilidades multiusuario (varios usuarios trabajando simultáneamente en el mismo ordenador) y multitarea (un usuario puede ejecutar varios programas simultáneamente) el papel de los CP/M y MS/DOS en los pequeños ordenadores monopuesto. Y, por ahora, el lenguaje C es el lenguaje de alto nivel que mejor aprovecha las posibilidades de estos equipos.

# CAPITULO II

## ELEMENTOS DEL LENGUAJE C




ntes de pasar a describir en detalle el C en este capítulo pasaremos una breve revisión a sus elementos: tipos de datos, operadores y estructuras de control que, aunque serán objeto de estudio en capítulos posteriores, se presentan aquí brevemente con la finalidad de dar una pequeña visión global del lenguaje y facilitar la comprensión de los ejemplos posteriores.

### Tipos de datos

El lenguaje C soporta los siguientes tipos de datos:

— Caracteres, ya sean ASCII o EBCDIC (según el juego de caracteres empleado).
— Enteros, con o sin signo.
— Números en coma flotante, en simple o doble precisión.
— Funciones.
— Arrays de cualquier tipo de variables.
— Punteros a cualquier tipo de variables.
— Estructuras.
— Uniones.

## *Operadores*

Existen los siguientes operadores:

— Aritméticos (+, -, *, /, %).
— Relacionales (>, >=, <, <=, ==, !=).
— Lógicos (&&, ||, !).
— Manejo de bits (&, |, ∧, <<, >>, ).
— Asignación (=, <op>=, ++, --).
— Condicional (? :).
— Acceso a datos (*, &, [], ., →).

## *Estructuras de control*

Las estructuras de control disponibles en lenguaje C son:

— Sentencias o expresiones.
— Bloques; segmentos de programa encerrados en dos llaves "{" y "}".
— Sentencia If-else.
— Sentencia switch.
— Bucles for, while, do y until.
— Saltos y etiquetas (goto, break, continue).

## *Tipos de constantes*

El C admite la definición y empleo de los siguientes tipos de constantes:

— **Enteros:** dígitos numéricos. Se interpretan como números decimales, a menos que su primer dígito sea un 0, en cuyo caso se interpretan como números expresados en octal. Por ejemplo, la constante 30 en decimal equivaldría a la constante 036 en base octal.

— Ox u OX indican un número en base hexadecimal (emplean los caracteres [0..9][a..f] o [0..9][A..F]. Por ejemplo, 30 puede escribirse como Ox1E o Ox1e.

Las letras "l" o "L" después de una constante entera indican que se trata de un "long integer", o entero largo. Por ejemplo 30l o 30L.

— **Números en coma flotante,** terminados en un "." o seguidos por una "e" o una "E". Por ejemplo, 6., 5.0, 123.456e-7, 0.987E6.

— **Caracteres.** Entre comillas simples ('). Por ejemplo 'a'.

Los caracteres de control o no gráficos se representan por secuencias de escape: \ n (retorno de carro), \ t (tabulador horizontal), \ r (salto de línea), \ 0 (carácter nulo, ASCII 0), \ (carácter "\"),



\' (comilla simple, "'"), o \ xxx, donde xxx representa un número octal de 3 dígitos.

— **Constantes alfanuméricas:** cero o más caracteres entre comillas dobles (").

## *Entrada/salida simple*

La función getchar() lee el siguiente carácter presente en la entrada (el fichero estándar de entrada es normalmente el teclado). Análogamente, putchar(c) escribe el carácter "c" a la salida (el fichero estándar de salida es normalmente la pantalla).

printf(formato, primero, segundo, tercero, ...) es una función de formateado de propósito general. Los caracteres que aparecen en el literal "formato" son enviados a la salida estándar, lo mismo que putchar, salvo que se encuentre un signo %, indicando que sigue una especificación de formato. Por ejemplo, %6.1f es semejante a F6.1 de Fortran, imprimiendo un número en coma flotante utilizando al menos 6 caracteres, con uno después del punto decimal.

%f imprime un número en coma flotante.

%d, %o y %x imprimen un número entero o un carácter en decimal, octal o hexadecimal, respectivamente, mientras que %D, %O y %X imprimirán un "long int" (o "entero largo", en doble palabra), en decimal, octal y hexadecimal (o también ld, lo y lx).

c imprime un único carácter. Así, printf("%c", c) es parecido a putchar(c).

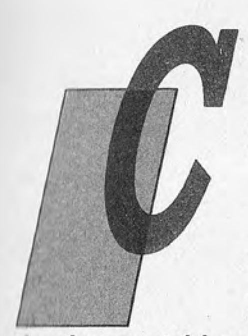
s imprime una cadena de caracteres (string), terminada por el código ASCII 0 (carácter nulo o terminador de string).

% imprimirá el carácter %.

Todas estas funciones forman parte de la denominada "librería C estándar de Entrada/Salida", pero no forman parte directamente del lenguaje C y serán vistas en mayor detalle al hablar de la librería C estándar.

# CAPITULO III

## EL COMPILADOR



C es un lenguaje compilado, a diferencia de otros, interpretados o seudocompilados. Vamos a describir un poco lo que es todo esto.

Los lenguajes compilados son traducidos a lenguaje de máquina por un programa especial que se denomina compilador. Este no es más que un programa que se limita a analizar nuestro programa fuente (el original que hemos escrito), comprobando su sintaxis, guardando unas tablas con las variables y direcciones que manejamos en él y transcribiéndolo al lenguaje interno del ordenador que estemos utilizando, obteniendo lo que se denomina "programa objeto".

El programa objeto ha de ser enlazado (linked) con una serie de subrutinas (como las utilizadas para gestión de memoria y periféricos) que le permitirán convertirse en un programa ejecutable, semejante a los restantes comandos de nuestro ordenador. En la figura 1 podemos observar las fases necesarias para proceder a la compilación de un programa.

Una vez que tenemos el módulo objeto será necesario "montarle" (linkado) las librerías necesarias para convertir el módulo en ejecutable.

Los programas interpretados no siguen esta secuencia de pasos, utilizando un módulo ejecutable (denominado run-time) que contiene las subrutinas necesarias para proceder a la ejecución del código fuente instrucción a instrucción.

En ocasiones se procede a un análisis sintáctico previo del código fuente; es una forma intermedia que facilita posteriormente la ejecución por parte del módulo de "run-time", hablándose en-

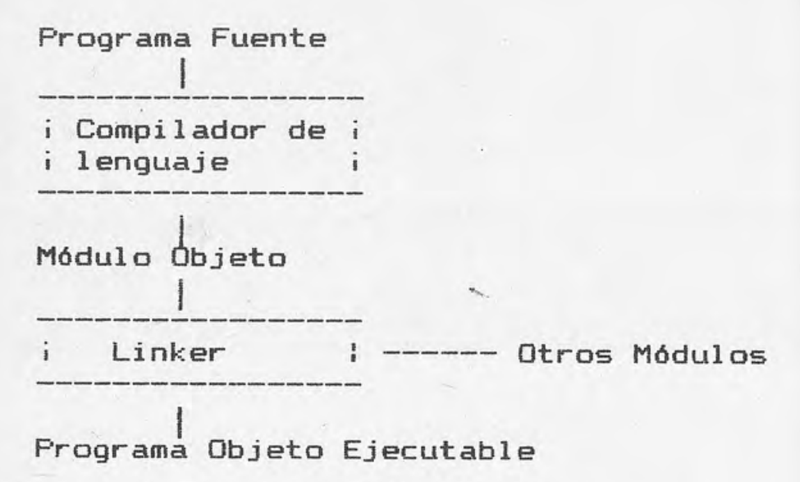



*Figura 1.—Fases de compilación de un programa.*

tonces de programas "seudo-compilados", a medio camino entre los programas interpretados y compilados; su estructura la podemos observar en la figura 2.

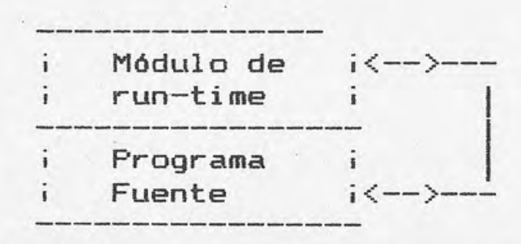



*Figura 2.—Noción de programa «seudo-compilado».*

Ejemplos típicos de lenguajes interpretados son el BASIC y el PASCAL, y de lenguajes seudo-compilados, el RM/COBOL y algunos BASIC existentes para microordenadores.

Un lenguaje compilado alcanza una velocidad de ejecución mucho mayor que la de los lenguajes interpretados o seudo-compilados, aunque ocupa más memoria al tener que incluir los módulos que convierten al programa en ejecutable.



En general, un programa compilado consta de tres zonas bien diferenciadas: de texto, datos inicializados y stack, como podemos observar en la figura 3. La zona de texto se refiere al código del programa; los datos inicializados, a las zonas fijas de memoria destinadas a contener datos que no van a ser modificados jamás, y el stack, al almacenamiento de variables temporales y retornos de funciones.




*Figura 3.—Aspecto interno de un programa compilado.*

Cuando la zona de texto de un programa compilado puede ser compartida por varios procesos que estén ejecutando simultáneamente el mismo programa, entonces se habla de un "texto puro" o de un "programa ejecutable puro", obteniéndose un importante ahorro de memoria central al no ser necesario almacenar varias copias del programa, sino una sola compartible por los distintos procesos que así lo requieran.

## El compilador de C

Para compilar programas en lenguaje C se emplea un programa denominado "cc" (c-compiler) utilizado como:

```
cc nombre.c
```

donde "nombre.c" representa un programa fuente escrito en lenguaje C; se utiliza normalmente la terminación ".c" para indicar este hecho.

El compilador de C buscará en la denominada "librería de C estándar" aquellas funciones de las que haga uso el programa

"nombre.c" y las montará sobre el mismo para obtener un programa ejecutable.

En la llamada al compilador se le pueden "pasar" (comunicar) opciones como el indicarle un nombre de salida (opción -o) para el programa ejecutable (si no tomaría por defecto el nombre de salida "a.out"). Así

cc -o nombre nombre.c,

compilaría el programa "nombre.c" denominando "nombre" al programa ejecutable. Otras opciones para el linker serían la obtención de un módulo ejecutable puro o el montaje con otras librerías distintas de la librería C estándar. Por ejemplo:

cc -O -nv -o nombre nombre.c -lm -lcurses -ltermcap

Sobre el sistema operativo Unix tendría el significado de compilar el programa fuente escrito en lenguaje C "nombre.c", llamando al programa resultante "nombre" (opción -o), empleando la posibilidad de optimización de código (opción -O), suministrando código ejecutable puro (opción -n), con indicación de las fases seguidas en la compilación (opción -v o "verbose") y con búsqueda de funciones en la librería matemática de C (opción -lm), en la librería "curses" (opción -lcurses) y en la librería "termcap" (terminal capabilities) con la opción -ltermcap.

La forma general de un programa en C que incluya el manejo de alguna función es la mostrada en la figura 4.

El ejemplo más sencillo de un programa en C podría ser:

```
main()
{
        printf("hello, world");
}
```

Denominamos a este programa "hola.c" y procedemos a compilarlo con la línea de comando

cc hola.c

Si todo ha ido bien y no nos ha faltado poner ninguna llave ("{" o "}"), ni el punto y coma ";", ni los paréntesis que flanquean a "main" ("main()") entonces, si ejecutamos el comando "a.out" obtendremos como respuesta el texto "hello,world" y habremos compilado y ejecutado con éxito nuestro primer programa en lenguaje C.

Este ejemplo está tomado del libro de Kernigham y Ritchie "El lenguaje de programación C" y es típico en la mayoría de los



```
#include <stdio.h>        -|  Inclusión de ficheros de texto
#define MS/DOS 3           |> y definición de símbolos mane-
...                       -|  jados por el preprocesador de
                              lenguaje C.

int j;                    -|  Definición de variables
double y;                  |> "globales" o comunes al
...                       -|   programa principal y a
                               las funciones.

main(argc, argv)          -|  Programa principal, con
int argc;                  |> argumentos en la línea de
char *argv[];             -|  llamada.
{
        int i;            -|  Variables locales, o
        char c[10];        |> pertenecientes sólo al
        float x;          -|  programa principal
        ...

        CUERPO DEL PROGRAMA PRINCIPAL
        Condiciones, bucles,
        llamadas a funciones, etc.
}


gosub(a, b, ..., n)       -|  Función con argumentos
int a;                     |> y definición de los
...                        |  mismos.
float n;                  -|


{
        int i;            -|  Variables "temporales"
        ...                |> o pertenecientes sólo
                          -|  a la función

        CUERPO DE LA FUNCION
        ...
}
```

*Figura 4.—Forma general de un programa C.*

libros como el programa más sencillo y el primero que debemos de compilar antes de pasar a ejemplos más complicados.

Ahora podríamos modificar el texto "hello,world" y convertirlo en "hello,world\n" de modo que se produjese un salto de línea después de imprimir el texto. (Nota: Se trata de la barra inclinada a la izquierda "\", no del signo habitual de división "/". Si nuestro teclado está configurado en castellano, en lugar de como ASCII USA, y pierde el símbolo, emplearemos el carácter "Ñ" en lugar de "\").

Podríamos seguir esta línea de explicación describiendo ejemplos cada vez más complicados en los que fuésemos incorporando sucesivamente los distintos elementos del lenguaje C, pero para eso ya están los libros incluidos en la bibliografía. Es preferible pasar una rápida revisión al lenguaje C y a sus elementos, poder detenerse un poco más en aspectos como las librerías de funciones, la escritura de alguna función de librería, y apuntar algún concepto que otro de C avanzado, que es lo que veremos en los próximos capítulos.



# CAPITULO IV

## EL C MAS A FONDO

### Nombres de variables

Una variable no es más que una posición de memoria donde almacenar un valor. Las variables se caracterizan por un nombre, un tipo de dato al que hace referencia y un tipo de almacenamiento, indicando de qué manera se lleva éste a cabo.

Los nombres de variables consisten en secuencias de letras y dígitos. Por ejemplo: cuenta, importe, j5, x, son posibles nombres de variables en lenguaje C.

El primer carácter debe ser siempre una letra, no un dígito (5j no es un nombre válido, a diferencia de j5, que sí lo es). El carácter especial de subrayado "_" se admite como una letra más, siendo muy utilizado en C como separador en nombres compuestos; nombres de variables válidos en otros lenguajes, como "x.max" o "saldo.debe" se expresarían en C como "x_max" y "saldo_debe". No se puede emplear un punto "." como separador en estos nombres compuestos porque este carácter posee un significado muy especial en lenguaje C, utilizándose para referenciar a los elementos simples integrantes de unos tipos complejos de datos denominados estructuras, de los que ya hablaremos en su momento. Según esto seguirían siendo válidos nombres de variables como "x_max", "saldo_debe", "xyz123_45" e incluso "_x".

En los nombres de variables se establece diferencia entre el empleo de letras mayúsculas y minúsculas. Los nombres "MAYOR", "Mayor" y "mayor" son todos diferentes. Como regla general, los nombres de variables propiamente dichas se referencian en minúsculas, reservándose las mayúsculas para los nombres ma-

nejados por el "preprocesador" de lenguaje C. De momento pensemos en que los nombres en mayúsculas se reservan para las "constantes" o valores que no pueden modificarse, a diferencia de las variables. Ya veremos que esta distinción es sólo aproximada, pero nos puede servir por ahora.

Los nombres de variables pueden tener en principio cualquier longitud, aunque sólo tienen sentido los ocho primeros caracteres. Según esto, nombres como "valor_de_x" o "valor_de_y" harían en realidad referencia a la misma variable "valor_de".

Si nombres de variables van a ser utilizados como referencias externas (es decir, si van a ser utilizadas por varios módulos o funciones de un programa), puede suceder que la longitud efectiva quede reducida a menos de ocho caracteres (normalmente siete). Esta reducción viene impuesta por el sistema operativo, que limita la longitud de los símbolos (nombres de variables) manejados por programas como el ensamblador y el montador de enlaces (linker).

Como en todos los lenguajes de programación existe una lista de palabras reservadas (keywords) que no pueden utilizarse como nombres de variables, pues el compilador no sería capaz de distinguir si realmente están haciendo referencia a una variable o a una instrucción. Estas palabras reservadas son:

| | | | |
|---|---|---|---|
| auto | else | int | switch |
| break | entry | long | typedef |
| case | enum | register | union |
| char | extern | return | unsigned |
| continue | float | short | until |
| default | for | sizeof | void |
| do | goto | static | while |
| double | if | struct | |

Como podemos observar se trata de una lista realmente breve en comparación con las existentes para otros lenguajes, como BASIC, COBOL y Pascal, en los que pueden existir hasta unas 150 palabras reservadas.

En consecuencia, podemos afirmar que un programa en lenguaje C se limita al empleo de este repertorio de instrucciones tan reducido y, por tanto, fácil de aprender. La complejidad residirá en manejar estas pocas instrucciones para obtener bloques de programa más elaborados (funciones) y su interconexión hasta lograr programas de mayor complejidad.



## Tipos de datos en C

Una variable siempre estará asociada a un tipo de dato en cualquier lenguaje de programación. El C soporta los siguientes tipos de datos:

- char: consiste en un único byte, capaz de almacenar un carácter correspondiente al juego de caracteres empleado, ya sea ASCII o EBCDIC.
- int: secuencia de dígitos correspondiente a un número entero, sin parte decimal, pudiendo tener signo o no.
- float: número en coma flotante y simple precisión.
- double: número en coma flotante y doble precisión.

El tipo char ocupa 8 bits y almacena valores en el rango de -128 a 127, utilizando un único byte de memoria; de manera más gráfica se ve en la figura 1.

```
 ------
| b0 |
 ------

-128 <= char <= 127
```

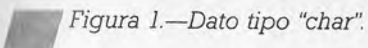


*Figura 1.—Dato tipo "char".*

Los valores del tipo int ocupan 16 bits, en direcciones contiguas, y contienen valores entre -32.768 y 32.767 almacenados en dos bytes de memoria (Fig. 2).

```
 -----------
| b0 | b1 |
 -----------

-32.768 <= int <= 32.767
```

*Figura 2.—Dato tipo "int".*

Los float ocupan 32 bits de memoria, estando formados internamente por un bit de signo, 8 bits de exponente (exponentes positivos y negativos) y una mantisa (parte fraccionaria) de 23 bits, proporcionando un máximo de 6 dígitos de precisión (Fig. 3).

```
|sg| 8 bits exp. | 23 bits mantisa (parte fraccionaria) |
```

```
| b0 | b1 | b2 | b3 |
```

*Figura 3.—Dato tipo "float".*

Los double ocupan 64 bits de memoria, distribuidos en un bit de signo, 8 bits de exponente y 55 bits de mantisa (parte fraccionaria), alcanzando 15 dígitos de precisión (Fig. 4).

```
|sg| 8 bits exp. | 55 bits mantisa (parte fraccionaria) |
```

```
| b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7 |
```

```
-0,000000000000001 <= mantisa double <= 999.999.999.999.999

            (15 dígitos de precisión)
```

*Figura 4.—Dato tipo "double".*



Al tipo int se le pueden aplicar los calificativos "short" y "long", obteniéndose los tipos de datos "short int" (entero corto) y "long int" (entero largo, o en doble palabra). La palabra int puede omitirse en estos casos, como normalmente se suele hacer. Short y long proporcionan enteros de diferente tamaño y precisión a los de int. Long se refiere a un doble entero (normalmente 32 bits), pudiendo almacenar valores comprendidos entre -2.147.483.648 y 2.147.483.647, mientras que short se refiere a un entero de 16 bits, con valores entre -32.768 y 32.767.

El tipo int sin calificativos puede referirse tanto a long como a short, dependiendo del ordenador considerado; por ello es preferible la utilización de short o de long para evitar problemas de portabilidad y pérdida de precisión al pasar los programas a otro ordenador.

El calificativo unsigned (sin signo) se puede aplicar a los tipos char e int (independientemente de short y long). El tipo unsigned hace que el bit de signo pierda su significado y que los números sólo puedan ser mayores o iguales que cero. Así el tipo "unsigned short" amplía la precisión de short al rango de valores entre 0 y 65.535, mientras que unsigned long puede contener valores entre 0 y 4.294.967.295.

## Tipos de almacenamiento

Además de un nombre y un tipo de dato, las variables también poseen un tipo de almacenamiento, consistente en una indicación al compilador de lenguaje C acerca del modo como debe asignarle memoria a esa variable.

- Almacenamiento auto (automático): indica al compilador que reserve una posición para la variable en la pila o stack. Esta posición de almacenamiento es temporal, de manera que cada vez que se produce una llamada a función y se define en ella una variable del tipo auto se emplea una nueva posición de memoria para la variable, pudiendo reutilizarse esa posición para contener cualquier otra variable después de que se haya producido el retorno de la función.
- Static (estático): produce la asignación de una posición fija (estática) de memoria, de manera que cada llamada a la función utiliza siempre la misma posición de memoria, conservando inalterable su contenido en el transcurso de dos llamadas consecutivas a la función.
- Extern (externo): es semejante a static, con la diferencia de permitir que otras funciones accedan a la misma variable, mientras que el otro sólo permite su uso por la función en que fue definida la variable.

- Register (registro): es semejante a auto. Indica al compilador que, a ser posible, emplee un registro interno de la CPU para almacenar la variable en lugar de una posición de la memoria central, para disminuir el tiempo de acceso a lás variables referenciadas muy frecuentemente. Los registros sólo pueden contener valores enteros, estando limitado el número de registros utilizables a la arquitectura de la CPU del ordenador. Muchos compiladores de lenguaje C admiten el tipo register por razones de compatibilidad, aunque luego internamente lo ignoren. No está permitido intentar acceder a la dirección de memoria de una variable de tipo register al no tratarse de una dirección fija, a diferencia de las direcciones de memoria central.

## Declaración de variables

En C, como en todos los lenguajes compilados, es necesario definir todas las variables de un programa (o función) antes de que puedan ser utilizadas. La declaración de variables consiste en un tipo de almacenamiento (opcional), seguido de un tipo de variable y de una lista de los nombres de variables a los que afecta. Por ejemplo:

```
auto int x, y, z;
register unsigned short int x1;
long john;
```

Las variables se pueden distribuir a lo largo de tantas declaraciones como se desee. Así, las anteriores declaraciones de variables también se podrían haber escrito repartidas en varias líneas:

```
auto int x;
auto int y;
auto int z;
```

Si no se ha especificado explícitamente ningún tipo de almacenamiento se supone por defecto que se trata del tipo extern (variable global, o común al programa principal y a todas las funciones), salvo que la declaración se haya realizado en el interior del cuerpo de una función, en cuyo caso se asume el tipo auto (variable local, o perteneciente sólo a la función).

Está permitido que en una declaración se inicialice una variable, asignándole un tipo de almacenamiento y un valor inicial en la misma instrucción, con sólo acompañar al nombre de la variable de un signo igual y una expresión. Por ejemplo:



```
int x = 10;
char asterisco = '*';
```

Definirían "x" como un entero, asignándole un valor inicial de 10, y "asterisco" como un char conteniendo un "*".

Si el tipo de almacenamiento es static o extern entonces la expresión tiene que ser una constante, debiendo realizarse la inicialización una sola vez antes de que el programa sea ejecutado.

Las variables de tipos automatic y register son inicializadas automáticamente por el compilador a cero (variables numéricas) o al carácter nulo (variables literales o strings).

## Conversiones de tipo

Cuando en una expresión se utilizan variables de tipos diferentes, el compilador se ve obligado a realizar una unificación de tipos antes de proceder al cálculo de la expresión. Las conversiones de tipo de variables implícitas en lenguaje C siguen el orden de precedencia mostrado en la figura 5.

```
double  <- float
  |
long int
  |
short int
  |
 int
  |
 char
```

*Figura 5.—Orden de precedencia de las variables en lenguaje C.*

Esto quiere decir que si, por ejemplo, en una expresión interviniesen variables de los tipos char e int, el tipo char se convertiría a int antes de proceder a evaluar la expresión. Análogamente, si los operandos fuesen int y double, se convertirían a double.

Esta regla se aplica no sólo a las variables y expresiones, sino también a los argumentos de funciones.

Para las expresiones en que sólo aparezcan los tipos int cuando sea posible se seguirá la regla de la figura 6.

```
unsigned int

      |
     int
```

Figura 6.—Orden de precedencia de las variables int.

La conversión de un número en coma flotante (float) a un entero se realiza por truncación. Para números positivos se toma el entero menor o igual al número en coma flotante; en cambio, para números negativos, C no especifica el sentido del truncamiento si se desprecian los decimales por defecto o por exceso. Esto también depende del ordenador, de modo que el fabricante elige la solución que le resulte más sencilla.

Es posible forzar una conversión de tipo en un momento dado, mediante el empleo del operador "cast", utilizando como "(nuevo tipo) expresión", así:

```
int x;
z = (long) x;

char s[10];
p = (char *) s;
```

forzaría la conversión no de la variable "x", pues "x" permanecería inalterado en su valor y tipo, sino de su valor al intervenir en una expresión, haciendo que "z" se convirtiese al valor de "x" considerado como si fuese del tipo long. Análogamente "p" se interpretaría como un puntero al array "s".

El operador cast se emplea con frecuencia para realizar conversiones de tipo en llamadas a funciones. Puede suceder que una función espere argumentos de llamada de un determinado tipo y que las variables que deseamos pasarle como argumentos no se correspondan con ese tipo; esto se soluciona con el empleo de cast.



## Tipos de operadores y expresiones de asignación

La figura 7 muestra los distintos tipos de operadores existentes en lenguaje C, que pasamos a describir con más detalle a continuación.

<table>
<tr><td>Aritméticos</td><td>(+, -, *, /, %)</td></tr>
<tr><td>Relacionales</td><td>(>, >=, <, <=, ==, !=)</td></tr>
<tr><td>Lógicos</td><td>(&&, ||, !)</td></tr>
<tr><td>Manejo de bits</td><td>(&, |, , <<, >>, )</td></tr>
<tr><td>Asignación</td><td>(=, <op>=, ++, --)</td></tr>
<tr><td>Condicional</td><td>(?:)</td></tr>
<tr><td>Acceso a datos</td><td>(*, &, [], ., ->)</td></tr>
</table>


Figura 7.—Operadores en Lenguaje C.

## Operadores aritméticos

Los operadores binarios son + (suma), - (resta), * (multiplicación), / (división) y % (resto módulo). También existe un operador unario - (cambio de signo).

El orden de prioridad de los operadores es como en matemáticas: *, / y % tienen la prioridad más alta, calculados de izquierda a derecha, y se ejecutan antes que el + y el - binarios. Algunos compiladores aplican la ley distributiva del producto respecto a la suma, calculando la expresión a*(b+c) como a*b+a*c, especialmente cuando "a" y "c" son constantes (como es frecuente en el caso de subíndices de arrays).

## Operadores lógicos y relacionales

El lenguaje C no tiene definidos tipos de datos especiales para las variables booleanas "cierto" y "falso", empleándose para

ellas valores enteros: distinto de cero para cierto (true) y cero para falso (false).

Los operadores de relación son > (mayor que), < (menor que), >= (mayor o igual) y <= (menor o igual).

Los operadores de comprobación de igualdad son == (igual) y != (distinto de). Un error frecuente entre los programadores en C consiste en intercambiar el operador de asignación (=) por el operador de comprobación de igualdad (==), realizando una asignación cuando lo que se deseaba realmente era una comparación entre dos variables o expresiones. Esto no siempre provocará un error del compilador, resultando en ocasiones difícil de detectar.

Los operadores conectivos lógicos son && (and) y || (or). Siempre se evalúan de izquierda a derecha, deteniéndose la evaluación tan pronto como se produzca el fallo de una condición (para el and) o se cumpla (para el or), sin necesidad de evaluar la totalidad de la expresión lógica.

La prioridad de ejecuciión más alta corresponde a los operadores relacionales (>, >=, < y <=) y a las conectivas lógicas (&& y ||).

## Operadores de manejo de bits

Los operadores de manejo de bits son & (and), | (or), ∧ (or exclusivo), << (desplazamiento a la izquierda) y >> (desplazamiento a la derecha). Estos operadores actúan a nivel de los bits de una variable y sólo pueden aplicarse a variables enteras, no en coma flotante. También existe un operador que proporciona el complemento a uno de una variable, cambiando los bits a 1 por 0, y viceversa.

Una aplicación típica de los operadores de manejo de bits consiste en su utilización para manejar flags (indicadores). Podemos tener una variable entera que almacene un conjunto de valores booleanos en forma de bits e inicializarlos y comprobarlos mediante el uso de máscaras conjuntamente con los operadores & y |. Por ejemplo, supongamos que ERROR es la máscara 0000001, entonces:

flags = flags | ERROR

pondría a 1 el último bit, correspondiente a ERROR, de flags.

if (flags & ERROR)

comprueba si flags tiene puesto el último bit (ERROR) a 1.

Hay que advertir de la posible confusión entre el empleo de los operadores de bit "&" y "|" y los operadores lógicos "&&" y "||".



## Operadores de incremento y decremento

El operador ++ añade 1 a su operando, que debe ser una variable, mientras que el operador -- le resta uno.

Estos operadores de incremento (++) y decremento (--) se pueden utilizar en los denominados modos pre-(incremento/decremento) o post-(incremento/decremento), conforme a la situación relativa del operador y la variable.

Si aparece antes el operador que la variable (modo pre), entonces se modifica la variable antes de hacer uso de su valor; por el contrario, si el operador aparece después que la variable (modo post), entonces ésta se verá modificada después de ser utilizada. Un ejemplo ayudará a comprender estos términos.

Consideremos la variable "x", con un valor inicial x = 5, y las expresiones siguientes:

| Operación | Funcionamiento | Resultado final y | Resultado final x |
|---|---|---|---|
| (1) y=x++ | Asigna a "y" el valor de "x" | 5 | 6 |
| (2) y=++x | Suma 1 a "x", asignándola a "y" | 6 | 6 |
| (3) y=x-- | Asigna a "y" el valor de "x" | 5 | 4 |
| (4) y=--x | Resta 1 a "x" asignándolo a "y" | 4 | 4 |

siendo:

(1) = post-incremento = x++
(2) = pre-incremento = ++x
(3) = post-decremento = x--
(4) = pre-decremento = --x

## Operadores de asignación y expresiones

El operador de asignación más simple es el igual "=", que no debe confundirse con el operador de comprobación de igualdad (==). Considerando las expresiones

```
if (a == b) {              if (x = y) {
     ...                        ...
}                          }
```

En la primera comparación (a==b) se obtendrá como resultado un uno (cierto) si "a" y "b" son iguales o un cero (falso) si son diferentes, sin que se vean modificados los valores de "a" y "b" por el hecho de comparar sus valores.

En el segundo caso (x=y) estamos asignando a la variable "x" el valor actual de "y", y si este valor es distinto de cero se toma como cierto en la expresión, con lo que estamos (inadvertidamente) modificando el valor de la variable "x".

El operando a la izquierda del signo igual debe ser una variable o un elemento de un array. Entonces el valor de la variable se reemplaza por el valor que aparezca a la derecha del signo igual, pudiendo ser tanto el resultado de una expresión como una constante o una variable. Así son válidas las siguientes asignaciones:

```
x = -1;          x = y * (x + 1);
```

También se pueden realizar asignaciones múltiples; si queremos asignar

```
x = 10;
y = 10;
z = 10;
```

se puede hacer como:

```
x = y = z = 10;
```

de modo que tanto "x", "y" como "z" se inicializarían al valor 10.

El valor resultante de una operación de asignación consiste en el nuevo valor de la variable situada a la izquierda del signo igual y es de su mismo tipo. Así, si "f" ha sido declarada del tipo float e "i" del tipo int, la expresión "f=i" es del tipo float, mientras que "i=f" es del tipo int, obteniéndose el resultado por truncamiento de la parte decimal de "f".

Pueden emplearse los operadores, de manera que expresiones como

i = i + 2;

también pueden escribirse como

i += 2;

Esto es válido para los operadores +, -, *, /, <<, >>, &, y |. En general, si "e1" es una variable, "op" un operador y "e2" una expresión, entonces

e1 op=e2;

es equivalente a

e1 = (e1) op (e2);

con la diferencia de que "e1" se calcula sólo una vez.



Si consideramos x = x + 1; se generará el siguiente código interno:

```
move A,x     ; Carga el valor de x en el acumulador
move B,1     ; Carga 1 en el registro B
add B        ; Suma B al acumulador
store x      ; Almacena el resultado en x
```

Mientras que la expresión x++ daría lugar a:

```
move A,x     ; Carga el valor de x en el acumulador
incr A       ; Suma 1 al contenido del acumulador
store x      ; Almacena el resultado en x
```

Observamos que en el segundo caso se economiza una instrucción, generándose un código más compacto y rápido.

Los operadores "++" y "--" equivalen a "+=1" y a "-=1", con la inclusión de los paréntesis necesarios. Por otro lado

x *= ++y;

es como escribir

x = y + 1;
x = x * y;

Los paréntesis también son significativos, así

x *= y + 1;

equivale a:

x = x * (y + 1);

Como ejemplo de manejo de operadores en C vamos a ver la función borra(s, c), que elimina todas las ocurrencias del carácter "c" en la cadena de caracteres "s".

```
/*
 * Función para eliminar el caracter(es) c
 * de la cadena de caracteres s
 */

borra(s, c)
char s[];
int c;
{
        int i, j;
        for (i = j = 0; s[i] != '\0'; i++)
                if (s[i] != c)
                        s[j++] = s[i];
                s[j] = '\0';
}
```

El funcionamiento del programa es el siguiente: mediante la instrucción for se explora el array "s" carácter a carácter. Cada vez que se encuentra un carácter distinto de "c" en la posición actual (i), éste es copiado en la posición "j" y se incrementa "j". Por último se añade el carácter nulo '\0' como terminador de string.

La instrucción s[j++]=s[i] sería equivalente a la secuencia:

```
s[j]=s[i];
j=j + 1;
```

Como aún no hemos mencionado la instrucción for ni el manejo de arrays, diremos brevemente que for se emplea para implementar bucles, constando de una expresión inicial (i = j = 0), una condición de final de bucle (s[i] != '\0'), que provoca la permanencia en el bucle mientras sea cierta la condición y hace que se ejecute la acción i++, así como el cuerpo (restantes instrucciones) del bucle.

Los arrays de caracteres s[i] y s[j] consisten en una colección de valores de tipo char, a los que se accede por medio de un subíndice ("i" o "j"). Así s[0] corresponde al primer elemento, s[1] al segundo, y así sucesivamente.

## Expresiones condicionales

Para calcular la mayor de dos variables "a" y "b" podemos escribir el siguiente programa:

```
if (a > b)
        z = a;
else
        z = b;
```

que asignaría a "z" el mayor valor de "a" o "b". La instrucción if incluye la comprobación de una condición lógica (a > b), ejecutándose la(s) instrucción(es) siguiente(s) al if en caso de que la expresión resulte cierta, y las instrucciones siguientes al "else" en caso de que la condición resulte falsa.

Otra forma de hacer lo mismo sería:

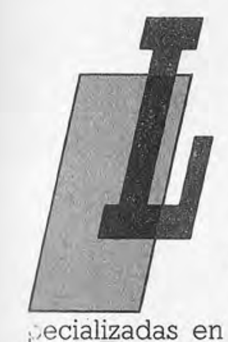
```
z = (a > b ? a b);
```

debido a que el compilador de C trata los signos "?" y ":" como si se tratase de una construcción if-else, de manera que si se verifica (a > b) asigna z = a (correspondiente a "?") y z = b en caso contrario (":").



# CAPITULO V

## ESTRUCTURAS DE CONTROL



Las estructuras de control son instrucciones (o conjuntos de instrucciones) que sirven para dirigir la ejecución de un programa, procediendo a la ejecución de unos u otros segmentos o módulos del mismo de acuerdo con el resultado obtenido tras la evaluación de determinadas condiciones lógicas. Para llevar a cabo la comprobación de estas condiciones lógicas existen las denominadas "instrucciones de control", especializadas en diversos tipos de comprobaciones y acciones subsiguientes.

Una sentencia consiste en una expresión seguida por un punto y coma ";", siendo normalmente una expresión de asignación, de incremento, decremento, o una llamada a función.

En C el punto y coma ";" es un carácter terminador de sentencia, en lugar de un separador como es el caso de Pascal. C requiere la utilización de más ";" que Pascal, pero su uso es más uniforme.

Consideremos como ejemplo el siguiente programa en C:

```
while (i < 10)  {
        j = i * i;
        printf("%d  d\n", i, j);
}
```

que podría escribirse en Pascal como:

```
while i < 10 do
begin
        j := i * i;
        writeln(i:4:0, j:4:0);
end
```

Las llaves "{" y "}" se utilizan para agrupar varias instrucciones en una sola sentencia o "bloque". Habitualmente se emplean para las sentencias if, else, while, do, until y for. Las llaves "{ }" son obligatorias en la declaración de una función.

El programa anterior imprimiría una tabla de 10 números, con sus valores y los de sus cuadrados. El control del programa se realiza mediante la instrucción while, que ejecuta una sentencia (el bloque entre llaves "{" y "}"), mientras se verifique una condición lógica ($i < 10$).

Pascal y lenguajes similares utilizan las palabras clave "begin" y "end", en lugar de "{" y "}", respectivamente.

## La sentencia IF

Las sentencias condicionales permiten la ejecución de una sentencia sólo en el caso de que se verifique una determinada condición lógica. La sentencia "if" permite tanto la ejecución si es cierta una condición, como la alternativa "else" para el caso de que no se verifique la condición. Su sintaxis es:

```
if (expresión)
        sentencia;
```

O también:

```
if (expresión)
        sentencia1;
else
        sentencia2;
```

Es obligatoria la utilización de paréntesis para separar la sentencia "if" de la expresión. En cambio no hace falta emplear la palabra "then" como en otros lenguajes.

Como "expresión" se puede utilizar cualquiera cuyo valor sea distinto de cero (cierto) o cero (falso). Así, es válida la siguiente condición:

```
if (s[i] != 0)
        c = s[i];
```

que también podría haberse escrito como:

```
if (s[i])
        c = s[i];
```



Como la sentencia "else" es opcional, un programa como

```
if ( n > 0)
        if (a > b)
                z = a;
        else
                z = b;
```

sería ambiguo, porque la sentencia "else" podría referirse a cualquiera de los dos "if". Así tanto podría significar:

```
if (n > 0) {
        if (a > b) {
                z = a;
        } else {
                z = b;
        }
} else {
                ;       /* sentencia nula */
}
```

como:

```
if (n > 0) {
        if (a > b) {
                z = a;
        } else {
                ; /* sentencia nula */
        }
} else {
        z = b;
}
```

Hemos colocado llaves "{ }" en todos los casos, incluidos los de "else" con sentencia nula, para distinguir las diferencias con mayor claridad.

El lenguaje C, como el Pascal y otros lenguajes, resuelve la ambigüedad asociando el "else" con la sentencia "if" más interior. Así el programa original es equivalente al primero, pero no al segundo.

Una construcción de sentencias if-else empleada con bastante frecuencia consiste en:

```
if (expresión 1)
    sentencia 1;
else if (expresión 2)
    sentencia 2;
```

```
else if (expresión n-1)
    sentencia n-1;
else
    sentencia n;
```

Esta construcción implementa una decisión alternativa entre varias elecciones mutuamente excluyentes entre sí. Básicamente la secuencia if-else se refiere a una decisión entre dos opciones, de manera que se irá evaluando la *"expresión i"* secuencialmente, ejecutándose la *"sentencia j"* para la primera "j" cuya *"expresión j"* sea cierta. Todas las *"expresiones j+1"* y *"sentencias j+1"* en adelante siguientes a "j" serán ignoradas. Sólo se llegará a ejecutar la *"sentencia n"* final si todas las expresiones anteriores son falsas.

## La sentencia SWITCH

La sentencia switch verifica una elección múltiple, comparando una expresión con cierto número de constantes de los tipos int o char. Así, por ejemplo

```
switch (c) {
        case ' ':
        case '\n':
        case '\t':
                n_blancos += 1;
                break;
        default:
                n_otros += 1;
                break;
}
```

La sintaxis utilizada es:

```
switch (expresión)
        sentencia;
```

Los paréntesis que aparecen en "(expresión)" son necesarios para separar "expresión" de la sentencia siguiente. Pascal utiliza la partícula "of" como separador.

Cualquier sentencia dentro de un switch puede estar precedida por una o más etiquetas "case" de la forma:

case constante-expresión:

donde cada una de las etiquetas "case" de la sentencia switch puede especificar distintas constantes.



Al final de la cadena de "case" se sitúa (opcionalmente) una sola etiqueta "default:", correspondiente al camino a seguir en caso de que no se haya verificado ninguno de los "case" anteriores.

La sentencia switch se ejecuta evaluando una expresión y comparándola secuencialmente con cada una de las alternativas "constante-expresión", especificadas por las etiquetas "case".

Si el valor de la expresión es igual a una de estas "constante-expresión", entonces la ejecución del programa se transfiere a la(s) instrucción(es) siguiente(s) a la sentencia etiquetada con el "case" en cuestión.

Dentro de la sentencia switch, la instrucción "break" provoca una salida inmediata del switch.

La sentencia switch es quizá más parecida al goto calculado de Fortran que al case de Pascal. Básicamente una instrucción switch es un goto a la etiqueta de un case determinado. A diferencia del goto calculado en Fortran, los case no tienen por qué comenzar en cero, o ser consecutivos. También a diferencia del case de Pascal una vez ejecutado un case de la sentencia switch el programa continuará su ejecución por el siguiente case a menos que se encuentre explícitamente una instrucción que altere esta secuencia (básicamente un brek, continue o return).

## Bucles WHILE

La sentencia while tiene como sintaxis:

```
while (expresión)
        sentencia;
```

evaluando en primer lugar la "expresión". Si ésta es cierta, entonces se ejecuta la "sentencia" una y otra vez, hasta que la expresión deje de ser cierta.

Si la expresión es falsa entonces no se ejecuta la sentencia y se transfiere el control a la instrucción siguiente a "sentencia".

Es obligatorio el empleo de paréntesis "()" para separar (expresión) de la siguiente sentencia. Pascal utiliza la palabra clave "do" para el mismo fin.

## Bucles FOR

La sentencia for:

```
for (expr1; expr2; expr3)
        sentencia;
```

es muy similar a la construcción:

```
{
        expr1;
        while (expr2) {
                sentencia;
                expr3;
        }
}
```

Los "()" y ";" que acompañan a expr1, expr2 y expr3 no son más que separadores. La equivalencia entre ambas construcciones for y while no es exacta; la única diferencia es que while admite el empleo de la sentencia "continue", que la hace diferente del bucle for.

La expresión "expr1" es la inicialización del bucle, "expr2" es la condición del bucle y "expr3" es la iteración; sentencia es la instrucción (o conjunto) a ejecutar tras cada paso.

Habitualmente, "expr1" y "expr3" son expresiones de asignación o funciones, mientras que "expr2" suele ser una expresión lógica o relacional. Tanto "expr1" como "expr2" y "expr3" son opcionales, pudiendo estar vacíos (o no existir), aunque los separadores ";" deben estar presentes en todos los casos. Así el bucle for más corto sería el bucle infinito:

```
for(;;)
        sentencia;
```

De este bucle sólo se podría salir mediante un salto como goto o return.

Los bucles for en lenguaje C son radicalmente diferentes de los bucles for de otros lenguajes. Por ejemplo, los bucles for en Pascal se restringen al control de una variable limitada a seguir un conjunto ascendente o descendente de valores, como:

```
for i := 0 to n - 1 do
        a[i] := 0
```

que tendría su equivalente en C de la forma:

```
for (i = 0; i < n; i++)
        a[i] = 0;
```

La analogía sólo es exacta en líneas generales, porque C permite que se modifiquen las variables "i" y "n" en el interior del cuerpo del bucle "for", mientras que esto no está permitido o es causa de error en Pascal.



Los componentes de la sentencia "for" son expresiones arbitrarias en lugar de constantes, esto les proporciona una mayor generalidad, permitiéndoles realizar iteraciones sobre cualquier progresión, no sólo sobre sucesiones aritméticas.

La ventaja de un bucle "for" sobre un "while" consiste en que en los bucles "for" las sentencias de control del bucle están agrupadas en la misma instrucción, lo que les confiere mayor claridad. Baste comparar el bucle "for" presentado anteriormente con

```
i = 0
while (i < n) {
        a[i] = 0;
        i++;
}
```

o su equivalente:

```
i = 0;
while (i < n)
        a[i++] = 0;
```

Esta diferencia es aún más notable en el caso de que el cuerpo del bucle "for" sea más extenso o si en su interior aparecen varias sentencias "if".

## Bucles DO-WHILE

Los bucles "for" y "while" siempre comprueban la condición de final de bucle antes de llegar a ejecutar el cuerpo del mismo. Sin embargo, en ocasiones se requiere que el cuerpo del bucle se ejecute al menos una vez y se compruebe luego la condición, como sucede en los bucles "do-while":

```
do
        sentencia
while (expresión);
```

```
{
        sentencia;
        while (expresión)
                sentencia;
}
```

Con la diferencia de que el código correspondiente a "sentencia" no necesita duplicarse.

Normalmente "sentencia" corresponde a una instrucción com-

puesta. Aunque se refiera a una instrucción simple se suelen incluir las llaves "{ " y " }" de principio y final por razones de claridad.
Un bucle como

```
do
        s[i++] = n % 10 + '0';
while ((n / 10) > 0);
```

es más difícil de ver que si se escribe como:

```
do {
        s[i++] = n % 10 + '0';
} while ((n / 10) > 0);
```

## Etiquetas y la sentencia GOTO

La sentencia "goto etiqueta;" produce que la ejecución del programa continúe en la instrucción que sigue a "etiqueta".

Una "etiqueta" consiste en un nombre de variable (una cadena de letras y dígitos comenzando por una letra), seguida por dos puntos ":".

Una "etiqueta" puede preceder a cualquier sentencia; todas las etiquetas de una función deben ser distintas entre sí.

Por supuesto, un "goto" debe referirse a una etiqueta de la misma función.

C no comprueba la existencia de "goto" sin sentido, como el salto al interior de un bucle sin pasar antes por la cabecera del mismo, esto es responsabilidad exclusiva del programador.

Las "etiquetas" tienen que ser constantes, no se permiten etiquetas variables.

Aunque es posible escribir programas utilizando sólo "goto" e "if" en la forma

```
if (expresión)
    goto etiqueta;
...
etiqueta: sentencia;
```

en lugar del empleo de "else", "switch", etc., es más fácil, limpio e incluso eficiente el empleo de las estructuras de control vistas anteriormente.



## GOTOs especiales: BREAK y CONTINUE

Los casos en que la utilización de "goto" es más eficiente y clara han sido resueltos mediante las instrucciones especiales "break" y "continue".

La sentencia "break" fuerza la salida inmediata de un bucle "for", "do" o "while", así como la sentencia "switch".

```
while (expresión) {
        ...
        break;
        ...
}
```

Es equivalente a:

```
while (expresión) {
        ...
        goto salida;
        ...
}
salida: sentencia;
```

La sentencia "continue" produce el salto a la siguiente iteración en un bucle "do", "for" o "while". Así:

```
while (expresión) {
        ...
        continue;
        ...
}
```

equivale a:

```
while (expresión) {
        ...
        goto sigue;
        ...
sigüe: sentencia;
}
```

Análogamente, para el bucle for:

```
for (expr1; expr2; expr3) {
        ...
        continue;
        ...
}
```

es equivalente a:

```
for (expr1; expr2; expr3) {
        ...
        goto sigue;
        ...
sigue: sentencia;
}
```

### Cuándo es necesario el empleo del GOTO

En ocasiones, es necesario salir de una estructura de bucles muy complicada realizada a varios niveles. Por ejemplo:

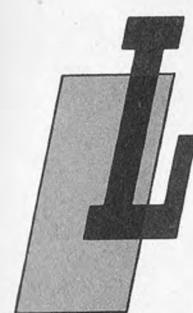
```
while ((c = getchar()) != EOF) {
        while (c == ' ' || c == '\n' ||c == '\t')
                if ((c = getchar()) == EOF)
                        goto salida;
        ...
}
salida: sentencia;
```

En este caso estaría justificado el empleo del goto por la dificultad de salir desde el bucle más interno hacia el exterior.



# CAPITULO VI

## FUNCIONES



Las funciones se emplean para dividir los programas grandes en otros más pequeños y manejables. Cada función consiste en una parte del programa, con sus propias variables, definiciones e incluso llamadas a otras funciones.

Los programas escritos en lenguaje C constan generalmente de una gran cantidad de pequeñas funciones de, a lo sumo, dos páginas de código fuente. Las funciones constituyen una manera natural de repartir un gran trabajo de programación entre un equipo de personas.

Una propiedad importante de las funciones es su posibilidad de trabajar con argumentos de llamada y devolver valores, pudiendo ser utilizadas en otros programas similares con sólo cambiar los datos que figuran en la llamada a la función.

### Definición de funciones

La definición de una función tiene la siguiente forma:

```
tipo nombre(lista-de-argumentos)
declaraciones-de-argumentos
{
        variables-locales;
        SENTENCIAS
        ...
}
```

"tipo" corresponde al tipo de dato devuelto por la función. Si no se ha especificado ninguno se supone que se trata del tipo int, por defecto.

La lista de argumentos consiste en una lista de nombres separados por comas. Esta lista puede estar vacía (función sin argumentos), pero incluso entonces siguen siendo necesarios los paréntesis "()" en la llamada a la función.

Las declaraciones de argumentos son las declaraciones (optativas) de las variables utilizadas como argumentos. Si no se especifica nada se supone que se trata del tipo int, aunque es buena práctica definir todos y cada uno de los argumentos para dar mayor claridad a la función y a los argumentos y tipos de datos que requiere.

Las variables locales son variables que tienen validez sólo en el interior del cuerpo de la función. Si los mismos nombres se utilizan en otras funciones o incluso en el programa principal (basta con que sea fuera de la función en que han sido definidas) entonces se referirían a variables diferentes.

El almacenamiento por defecto para estas variables es el tipo auto, de modo que cada vez que se llama a la función se vuelven a crear automáticamente todas sus variables locales. Recordemos que una variable es tan sólo el lugar para almacenar un valor, mientras que un nombre es una manera de referirse a una variable.

Dentro de una subrutina, la sentencia

return expresión

devuelve inmediatamente el resultado de la expresión como el valor devuelto por la función.

En C, a diferencia de Pascal y PL/1, no está permitido declarar unas funciones dentro de otras.

Si las funciones van a devolver un tipo distinto de int, debe declararse el tipo en la declaración de función de la forma:

tipo nombre-de-función ();

Al declarar el tipo de la función, la lista de argumentos ha de estar vacía "()" y se debe poner un punto y coma ";" al final. Si una función no se declara es porque se supone que devuelve un entero, como es el caso de las funciones empleadas habitualmente como printf y getchar, que se declaran implícitamente a través de su uso.

## *Llamadas a funciones*

Antes comentamos que la lista de argumentos podía estar vacía "()". Cuando se llama a una función los nombres de variables



que aparecen en la lista de argumentos son inicializados a los valores usados en la llamada a la función; así en el programa:

```
/* Eleva x a la n-ésima potencia
 * siendo x >= 0
 */
power(x, n)
{
        int i, p;
        p = 1;
        for (i = 1; i <= n; i++)
                p = p * x;
        return(p);
}

main()
{
        int i;
        i = power(2, 5);
        printf("i =  d\n", i);
}
```

La ejecución de la llamada a la función power como power(2, 5) crea dos nuevas variables con los nombres "x" y "n", que sólo son accesibles por la función power. La variable "x" se inicializa a 2, y la variable "n" a 5. Además, se crean dos nuevas variables "i" y "p", también accesibles sólo por la función y que no son inicializadas a ningún valor.

Las sentencias en el cuerpo de la función power se ejecutarán hasta llegar a la sentencia return(p). Esta sentencia causará que el valor de la función power(5, 2) sea el valor de la variable "p" (en este caso, 32). La ejecución del programa prosigue entonces en la instrucción de main posterior a la llamada a la función power (en nuestro caso I = power(5, 2)). Las variables creadas para efectuar el traspaso de argumentos a la función, así como sus variables locales, ya no son necesarias y sus posiciones de memoria pueden ser reutilizadas por otras variables y funciones.

# CAPITULO VII

## PREPROCESADOR MACROS. TIPOS DEFINIBLES Y CAMBIOS RECIENTES

### El preprocesador de lenguaje C

El preprocesador de lenguaje C consiste en un sencillo procesador de texto que realiza funciones de inclusión de ficheros, sustitución de macros e inclusión condicional de texto, previamente a la compilación de un programa.

Todas las sentencias del preprocesador comienzan con el carácter "#".

La sentencia del preprocesador empleada con más frecuencia es "#include", que realiza la inclusión de un fichero de texto dentro de un programa, pudiendo manejarse de dos formas:

- #include "nombre-de-fichero" incluye el contenido de un fichero dentro de un programa, realizando la búsqueda del fichero en el directorio original donde se encuentra el programa. Si no lo encontrase allí, entonces buscaría en una serie de directorios estándar, dependiendo del sistema operativo. (En el caso de Unix buscaría en el directorio "/usr/include").
- #include <nombre de fichero> es similar a la anterior, con la única diferencia de que el preprocesador sólo buscaría en los directorios estándar. Esta forma de #include se emplea habitualmente para incluir declaraciones y definiciones dependientes del sistema o instalación informática.

### *Sustitución de macros*

Las definiciones de la forma

#define nombre texto

causan que todas las ocurrencias de "nombre" sean reemplazadas por "texto". Nombre sigue las mismas reglas que las variables en C, estando formado por combinaciones de letras y dígitos. La sustitución de texto no se realiza en el interior de literales entre comillas (""), así si FALSO es un nombre que figura en un #define, no sería sustituido en printf("FALSO"). Esto es lo que se denomina una macro y "nombre" es, precisamente, su nombre.

Las macros pueden utilizar argumentos si en su definición se acompaña un paréntesis de apertura "(" a continuación de "nombre", reemplazándose los parámetros actuales de llamada por los que figuran en "texto". Si escribimos:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
x = max(p+q, r+s);
```

se convertiría en

X = ((p + q) > ? (p + q) : (r + s));

donde observamos que cada una de las expresiones (p+q) y (r+s) se evalúan dos veces. Esto podría causar problemas si una expresión viene acompañada de "efectos laterales" (side effects) indeseados, por ello la definición de "max" incluye paréntesis adicionales, ya que el preprocesador simplemente sustituye el texto, sin atender a otras consideraciones.

Un ejemplo de efecto lateral podría ser:

```
#define cuadrado(x) = x * x
y = cuadrado(z + 1);
```

que se convertiría en:

```
y = z + 1 * z + 1;
```

con un resultado final (y=2*z+1) muy distinto del deseado (y=(z+1)*(z+1)), como podemos observar.



### *Inclusión condicional de texto*

Una línea de control como

#if constante-expresión

verifica si la constante-expresión evaluada es distinta de cero.

#ifdef identificador

comprobaría si el identificador está "correctamente" definido en el preprocesador, es decir, si previamente ha aparecido en un #define.

#ifndef identificador

chequea si el identificador no está definido actualmente en el procesador.

Las tres formas descritas van acompañadas por un texto arbitrario y la línea de control

#endif

El texto puede contener adicionalmente una línea de control de la forma:

#else

Si la condición a comprobar es cierta, entonces se ignoran todas las líneas comprendidas entre #else (si está presente) y #endif. Por el contrario, si la condición es falsa, cualquier texto entre la condición y #else (si existe) o #endif (si no hay #else) será ignorada.

Por convención, el preprocesador define varios identificadores dependientes del sistema y de la instalación puestos a 1. Por ejemplo, bajo Unix funcionando en un PDP-11, el preprocesador define los identificadores unix y pdp11. Análogamente, un preprocesador para VAX/VMS deberá definir vax y vms.

Esto permite al programador la escritura de programas portables incluyendo varias alternativas dependientes de la máquina o del sistema. Por ejemplo:

```
/* Modo de leer una variable long de un
 * fichero binario, independientemente del
 * ordenador considerado.
 */
long x = 0;
```

```
#ifdef pdp11
        x |= (long)getchar() << 16;
        x |= (long)getchar() << 24;
        x |= (long)getchar();
        x |= (long)getchar() << 8;
#endif
#if vax
        x |= getchar();
        x |= getchar() << 8;
        x |= getchar() << 16;
        x |= getchar() << 24;
#endif
```

En el primer caso, el PDP-11 empaqueta un long como se muestra en la figura 1 (a) y en el segundo el VAX lo hace como en la figura 1 (b).

```
(a) | byte 2 | byte 3 | byte 0 | byte 1 |

(b) | byte 0 | byte 1 | byte 2 | byte 3 |
```

*Figura 1.—Formas de empaquetar una variable long. a) PDP-II b) VAX.*

Consiguiendo nuestra función desempaquetar una variable de tipo long en cualquiera de los dos sistemas considerados mediante la utilización de #defines combinados con las oportunas máscaras de bit.

## Tipos definibles

C permite la creación de nuevos nombres de tipos de variables. Precediendo una declaración por la palabra clave typedef, el nombre que figura en la declaración se considera como un nuevo nombre de tipo en lugar de como una variable. Por ejemplo:

typedef int LONGITUD;



hace que LONGITUD sea un sinónimo de int. La única limitación estriba en que

long LONGITUD metros;

no es una construcción válida, al no poderse añadir el calificativo long al typedef, a diferencia de

long int metros;

que sí es válida. Un ejemplo más útil podría ser:

```
typedef char *string;

main(argc, argv)
int argc;
string *argv;
{
        ...
}
```

y un ejemplo algo más complicado:

```
typedef struct _tnode {
        string word;
        int count;
        struct _tnode *left;
        struct _tnode *right;
} tnode;
```

Aunque tiene sus limitaciones, pues

```
typedef struct {
        string word;
        int count;
        tnode *left;
        tnode *right;
} tnode;
```

no funcionaría, al aparecer tnode en la definición del typedef y formando parte de sus componentes.

Typedef no define ningún tipo nuevo de datos, sólo cambia de nombre a los tipos de datos. Una variable declarada como del tipo "string" sería lo mismo que si se hubiera definido del tipo "char *", y el compilador de lenguaje C no establecería diferencia entre ellas.

Los tipos definibles pueden ser muy útiles para escribir fácilmente declaraciones de tipos complicados, difíciles de comprender, como puede ser el caso de:

typedef int (* PFI)();

Crea el tipo PFI como un "puntero a una función que devuelve un entero". Comparemos las expresiones

PFI strcmp, numcmp, swap;

con

int (* strcmp)(), (* numcmp)(), (* swap)();

Podemos apreciar la ventaja de introducir un tipo definible, obteniendo una expresión más breve y fácil de comprender.

Los tipos definibles suelen emplearse para facilitar la portabilidad. Por ejemplo, si una variable necesita ser declarada como un int en una determinada máquina y como long en otra, las líneas de control del preprocesador y los typedefs pueden emplearse para definir un nombre que pueda funcionar adecuadamente en cada una de las máquinas.

Aunque, en la práctica, si una variable ha de ser definida como long en alguna máquina sería razón suficiente para declararla como long en todas las demás.

## Cambios recientes en C

Algunos compiladores recientes de C reconocen la palabra "void" como un tipo sin valor alguno. Evidentemente, no se pueden declarar variables del tipo void, pero sí declarar que una función devuelve un tipo void para aquellos casos de funciones que no devuelven ningún valor (si el compilador se encontrase posteriormente un return, generaría un error).

Análogamente, se puede emplear un cast con los tipos void, para indicar que deliberadamente estamos definiendo una función que no devuelve valores. Por ejemplo:

(void) push(pop() + pop());

definiría la función push como una función que no devuelve valores.

C admite también los tipos numerados, semejantes a los que existen en Pascal. Un ejemplo podría ser:

enum color {rojo, verde, azul};

La sintaxis y el empleo de los tipos numerados son similares a los de las estructuras, con la diferencia de que no pueden establecerse miembros.



El primer elemento de un tipo numerado es equivalente a la constante 0, el siguiente a 1, y así sucesivamente.

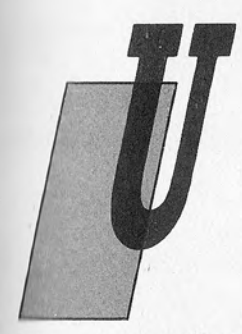
A un elemento de un tipo numerado se le puede asignar un determinado valor acompañando al nombre del mismo por un signo igual "=" y una expresión constante. Los elementos siguientes continuarán la sucesión de valores a partir del primer elemento asignado. Por ejemplo:

```
enum keywords {
        IF = 0200,
        ELSE,
        WHILE,
        UNTIL,
        DO,
        ...
};
```

Los tipos de datos numerados y sus elementos deben poder distinguirse entre sí, así como de las restantes variables "normales" y nombres de funciones (igual que la diferencia que debe haber entre las estructuras y los nombres de sus miembros).

# CAPITULO VIII

## *PUNTEROS Y ARRAYS*



Un puntero consiste en una pseudo-variable que dará la dirección de la variable asociada. Los punteros (como los goto) son una instrucción de muy bajo nivel, y puede ser fácil llegar a abusar de su empleo, pudiendo casi siempre ser sustituidos por otras construcciones alternativas, como arrays, y llamadas por referencia.

El operador & proporciona un puntero a una variable. Por ejemplo, si "x" es una variable, entonces &x es un puntero a "x".

El operador unario * accede a la variable apuntada por un puntero, así:

```
int x, y;
y = 10;
x = * ( &y) + 4;
```

es un equivalente un poco oscuro de:

```
int x, y;
y = 10;
x = y + 4;
```

Para declarar una variable (px) que contenga un puntero a un entero se hace:

```
int *px;
```

El aspecto de la declaración de px explica de alguna manera su utilización. La expresión *px apunta a un valor del tipo int.

Del mismo modo a como hemos definido px como un punte-

ro a un entero, podrían definirse punteros a otros tipos de variables, así:

double *dp;

Las expresiones *px y *dp tienen valores de los tipos int y double, respectivamente, y pueden ser utilizadas en otras expresiones.

El siguiente programa, una vez compilado y ejecutado, nos mostrará con valores numéricos el resultado de la utilización de los operadores * y &:

```
main()
{
        int x, *px;
        double *dp, d;
        px = &x;
        dp = &d;
        x = 1;
        d = 2;
        printf("*px = %d x = %d\n", *px, x);
        printf("*dp = %d d = %f\n", *dp, d);
}
```

Los operadores & y * tienen mayor prioridad de ejecución que los operadores aritméticos, así:

y = *px + 1;

toma el valor de la variable apuntada por px, le suma uno y asigna este valor a "y".

Como un puntero consiste en la dirección de alguna variable, una expresión precedida por un * también será una variable, de modo que también podrá intervenir en expresiones de asignación. Por ejemplo:

```
int x, *px;
x = 0;
px = &x;
*px = 1;
printf("x = %d *px = %d\n", x, px);
```

Las expresiones con un * pueden también utilizarse con operadores de incremento y decremento. Así obtendríamos expresiones como:

```
*px += 1;
++*px;
(*px)++;
```

que suma "1" al valor apuntado por "px"



## Punteros y argumentos de funciones

C pasa los argumentos a funciones utilizando "llamadas por valor", de forma que no hay manera directa, tras una llamada a una función, de que ésta pueda modificar las variables que le han sido pasadas como parámetros. En lenguajes como Fortran o Pascal, que utilizan "llamadas por referencia", una función sí puede modificar los valores de sus argumentos de llamada.

Consideremos la función swap, codificada en Pascal, que intercambia los valores de dos variables:

```
procedure swap(var x, y : integer);
    var
        temp : integer;
    begin
        temp := x;
        x := y;
        y := temp;
    end
```

Y la misma función codificada en Fortran:

```
SUBROUTINE SWAP(X, Y)
INTEGER X, Y, TEMP
TEMP = X
X = Y
Y = TEMP
RETURN
END
```

Si ahora transcribimos literalmente esta función a lenguaje C, obtendremos una función de aspecto parecido, pero totalmente inefectiva:

```
swap(x, y)
int x, y;
{
        int temp;

        temp = x;
        x = y;
        y = temp;
}
```

En lugar de esta versión podríamos escribir una función en C que intercambiase los valores de dos variables pasando como argumentos punteros a las variables en lugar de las propias variables:

```
swap(px, py)
int *px, *py;
{
        int temp;
```

```
        temp = *px;
        *px = *py;
        *py = temp;
}
```

y probarla mediante el siguiente programa principal:

```
main()
{
        int a, b;
        a = 1;
        b = 2;
        printf("a = %d b = %d\n", a, b);
        swap(&a, &b);
        printf("a = %d b = %d\n", a, b);
        return(0);
}
```

De hecho, los lenguajes de programación que pasan "por referencia" sus argumentos a una función en vez de "por valor", esencialmente lo que hacen es insertar un & antes de las variables en la lista de argumentos, y tratar los argumentos de la función como si fuesen punteros a variables, precediendo siempre con un * las operaciones en el interior del cuerpo de la función.

La llamada por referencia se emplea a menudo en funciones que realizan "retornos múltiples", es decir, funciones que necesitan devolver una indicación de error además de un valor, modificando variables globales del programa que llamó a la función.

## *Arrays*

Un array es un conjunto de variables del mismo tipo a las que se puede acceder directamente. Por ejemplo:

int a[10];

declara "a" como un array de 10 variables de tipo int. El índice del array empieza siempre en 0. Así el primer elemento de "a" es a[0], el segundo a[1], y así sucesivamente hasta a[9].

La expresión utilizada para el índice de un array es una expresión arbitraria, tanto constante como variable, del tipo int.

El acceso a todos los elementos de un array se realiza habitualmente por medio de un bucle for. Por ejemplo, para imprimir cada elemento del array definido anteriormente se podría utilizar la siguiente sentencia:

```
for (i = 0; i < 10; i++)
        printf("%d\n", a[i]);
```



C implementa internamente los arrays asignando a sus elementos posiciones consecutivas de memoria, tratando el array como si fuese un puntero a la primera variable del mismo. En el contexto del ejemplo anterior, C definiría el valor de la expresión a+i lo mismo que si fuese &a[i].

Esta operación de sumar un entero a un puntero se puede generalizar a punteros arbitrarios y no sólo a los punteros que apuntan al principio de un array. De hecho, C define una expresión de la forma a[i] como idéntica a "*(a+i)".

C define la sustracción entre dos punteros (llamémosles p1 y p2), del mismo modo que p2 es igual a

&p1[p1 - p2];

Por ejemplo, p1 - p2 es el número de elementos que hay entre p1 y p2. Si se restan dos punteros que no apunten al mismo array, el resultado obtenido será en general impredecible.

También pueden realizarse comparaciones entre punteros. Sin embargo, a menos que los punteros apunten al mismo array, sólo pueden realizarse comparaciones de igualdad (== y !=).

Como C no pone ningún cuidado en comprobar el rango de los índices que maneja es relativamente fácil "pasarse de una variable", manejando arrays y punteros, e intentar alcanzar una dirección no permitida, pudiendo llegar a invadir la zona de almacenamiento destinada a otras variables. Sin embargo, esta ausencia de chequeos tiene sus ventajas en cuanto a la sencillez de realización de un compilador de C y la posibilidad de trabajar con direcciones de máquina a bajo nivel.

A continuación vamos a presentar varias funciones de la librería de C estándar escritas de dos maneras diferentes: empleando arrays y empleando punteros.

```
/* Función strcpy: copia el literal s2
 * sobre el literal s1
 */
strcpy(s1, s2)
char s1[], s2[];
{
        int i;
        for (i = 0; s1[i] = s2[i]; i++)
                ;
}
```

Funcionamiento: va copiando s2 sobre s1 carácter a carácter mientras s2[i] sea distinto de cero (nulo).

```
/* Función strlen: devuelve la longitud
 * del literal s (calculada como el número
```

```
  * de caracteres distintos de nulo)
  */
strlen(s)
char s[];
{
        int n;
        for (n = 0; s[n]; n++)
                ;
        return(n);
}
```

Funcionamiento: mientras s[n] sea distinto de cero (nulo) va incrementando la variable "n", igual a la longitud del string.

```
/* Función strcmp: compara dos literales
 * s1 y s2, devolviendo >0 si s1<s2, 0 si
 * s1=s2, y <0 si s1<s2
 */
strcmp(s1, s2)
char s1[], s2[];
{
        int i = 0;
        while (s1[i] = s2[i])
                if (s1[i++] == '\0')
                        return(0);
        return(s1[i] - s2[i]);
}
```

Funcionamiento: va comparando s1 y s2 carácter a carácter. Mientras van coincidiendo los caracteres comprueba si se ha alcanzado el final de s1, en cuyo caso las cadenas s1 y s2 serían iguales, devolviendo un 0 la función. Si no es nulo se devolverá un valor >0 si es s1[i]>s2[i] o <0 si s1[1]<s2[i].

Estas mismas funciones se podrían haber escrito empleando punteros en vez de arrays, como:

```
strcpy(s1, s2)
char *s1, *s2;
{
        while (*s1++ = *s2++);
}
```

Funcionamiento: mientras s2 apunte a un carácter distinto de nulo, copia el valor apuntado por s2 sobre el valor apuntado por s1, e incrementa ambos punteros (++).

```
strlen(s)
char *s;
{
        char *p = s;
        while (*p)
                p++;
        return(p-s);
}
```



Funcionamiento: inicializa un puntero al principio del string. Mientras este puntero no apunte a un nulo lo va incrementando y al final retorna p-s que es el número de caracteres avanzados por el puntero y, por tanto, la longitud del string.

```
strcmp(s1, s2)
char *s1, *s2;
{
        for (; *s1 == *s2; s2++)
                if (*s1++ == '\0')
                        return(0);
        return(*s1 - *s2);
}
```

Funcionamiento: mientras los valores apuntados por s1 y s2 sean iguales, va desplazando ambos punteros, comprobando en cada momento si se alcanza el final de s1, en cuyo caso ambas cadenas serían iguales y la función devolvería un 0. Si s1 no apunta a un nulo, se devuelve la diferencia entre los valores apuntados por s1 y s2.

## *Arrays multidimensionales*

C permite la definición de arrays de cualquier tipo, no sólo de los tipos básicos.

En particular, los arrays multidimensionales son justamente arrays de arrays. Por ejemplo:

double a[5][5];

define "a" como un array de 5 × 5 elementos en coma flotante y doble precisión. Esta declaración es semejante a

```
var
    a : array[0..4,0..4] of real;
```

en Pascal, o a

DOUBLE A(5,5)

en Fortran.

En C un array de dos dimensiones se ve como un array de arrays, al igual que en Pascal. Es más, a[i] es un array de 5 elementos double. Sin embargo, a diferencia del Pascal, no puede abreviarse el doble subíndice.

```
double a[5][5];
int i, j;
for (i = 0; i < 5; i++)
        for (j = 0; j < 5; j++)
                a[i][j] = 0;
```

Este programa serviría para inicializar a cero todos los elementos del array "a", mostrando el manejo de subíndices de arrays.

## *Arrays como argumentos de funciones*

C convierte siempre una referencia a un array en un puntero al primer elemento del array. El compilador de C trata los argumentos declarados como arrays lo mismo que si hubiesen sido declarados como punteros al elemento base del array. Así:

```
strlen(s)
char s[];
{
        ...
}
```

es idéntico a

```
strlen(s)
char *s;
{
        ...
}
```

Declarar los argumentos como arrays o punteros es más bien una cuestión de estilo, aunque es recomendable declarar el argumento como un array si la función maneja índices de arrays, o como un puntero si accede al parámetro por medio de un *.

El límite de dimensión más a la izquierda de un array puede estar vacío en la declaración de un array como argumento. Supongamos que queremos pasar el array "pantalla" como un argumento a la función "f", que debería definirse como:

```
f(pantalla)
char pantalla[LINEAS][COLUMNAS];
{
        ...
}
```

o

```
f(pantalla)
char pantalla[][COLUMNAS];
{
        ...
}
```



o incluso como

```
f(pantalla)
char (*pantalla)[COLUMNAS];
{
        ...
}
```

La última declaración establece explícitamente que "pantalla" es un puntero a un array de tantos caracteres como COLUMNAS.

Hay que destacar que la función "f" debe tomar el número de columnas de la matriz pantalla, pero no necesita conocer el número de filas. C es más flexible que Pascal (en él hay que declarar todos los límites de un array pasado como argumentoo de una función), pero menos que Fortran (solamente necesita declararse la dimensión, por ejemplo, "pantalla" es un array de dos dimensiones).

## *Comparación entre arrays de punteros y arrays multidimensionales*

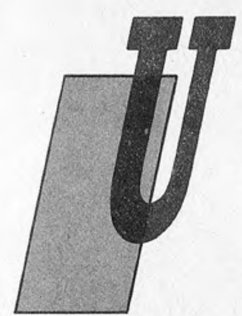
Existe una diferencia importante entre un array de punteros y un array bidimensional. Por ejemplo, dadas las declaraciones

```
char *lineptr[LINEAS];
char lines[LINEAS][MAXIMO];
```

El empleo de lineptr y lines parece similar en que lineptr[1][0] y lines[1][0] referencian ambos a un único carácter (presumiblemente el primer carácter de la segunda de un grupo de líneas de texto). Sin embargo, lines es un auténtico array, conteniendo LINEAS * MAXIMO caracteres. Como lines[i] es un array de MAXIMO caracteres, se accede a lines[i][j] multiplicando "j" por MAXIMO y sumándole "i" para determinar a cuál de los caracteres LINEAS * MAXIMO debería acceder.

# CAPITULO IX

## ESTRUCTURAS



na estructura es un conjunto de variables de los mismos o diferentes tipos. Las estructuras son semejantes a los registros (records) de Pascal.

Los elementos integrantes de una estructura se denominan miembros. Pascal llama campos (fields) a los elementos de una estructura, pero C reserva este nombre para referirse a los miembros particulares de estructuras que permiten manejo de bits.

La declaración de una estructura se hace como:

```
struct x {
        miembros-y-declaraciones
}
```

utilizándose como un tipo de dato más en declaraciones de tipo de variables.

"x" es el nombre de la estructura. Las referencias posteriores a la misma estructura pueden omitir las llaves ({}) y la declaración de sus miembros.

Los miembros y declaraciones incluidos en la estructura son como cualquier declaración de variable de las vistas hasta ahora, con la única diferencia de que no puede especificarse un tipo de almacenamiento (auto, static, register, extern). Por ejemplo:

```
struct fecha {
        int dia;
        int mes;
        int año;
};
```

```
struct persona {
        char nombre[TAMAÑO1];
        char direccion[TAMAÑO2];
        long cod_postal;
        long ss_num;
        double sueldo;
        struct date nacim_fecha;
        struct date incorp_fecha;
};
```

sería el equivalente a los records de Pascal:

```
type
    date = record
        dia: 1..31;
        mes: 1..12;
        año: integer;
    end;

    person = record
        nombre: array[1..TAMAÑO1] of char;
        direccion: array[1..TAMAÑO2] of char;
        cod_postal: integer;
        ss_num: integer;
        sueldo: real;
        nacim_fecha: date;
        incorp_fecha: date;
    end;
```

En este ejemplo definimos una estructura "fecha" conteniendo tres variables enteras: día, mes y año, y definimos una estructura "persona", conteniendo dos arrays de caracteres (nombre y dirección) cod_postal de tipo long, ss_num de tipo long, sueldo de tipo double, y dos miembros consistentes en estructuras "fecha", correspondientes a nacim_fecha e incorp_fecha.

Para acceder a un miembro de una estructura se utiliza el operador ".", como ya anticipábamos al hablar de las variables en C y sus tipos. Se recurre a él en la forma:

estructura-variable.miembro

Por ejemplo, para declarar una variable "d" como una estructura "fecha" e inicializarla a la fecha 1 de junio de 1986, se haría del modo siguiente:

```
struct fecha d;
d.dia = 1
d.mes = 6;
d.año = 1986;
```

Como podemos observar en la declaración de la estructura persona las estructuras se pueden encadenar, pudiendo utilizarse



como miembros de otras estructuras. Si se define "emp" como una estructura persona:

struct persona emp;

entonces

emp.nacim_fecha.año

se referirá al año de nacimiento del empleado en cuestión.

Una estructura externa o estática se puede inicializar acompañando el nombre de la estructura por una lista de inicializadores rodeados por llaves ({}).

struct fecha d = {1, 6, 1986};

Declara "d" como una estructura fecha, e inicializa d.día a 1, d.mes a 6 y d.año a 1986.

## Operaciones sobre estructuras

En implementaciones antiguas de C las únicas operaciones permitidas sobre las estructuras eran el acceso a sus miembros (mediante un "."), o el acceso a su dirección (mediante un &).

Los compiladores más recientes permiten la asignación de variables estructura, incluyendo el paso de estructuras como parámetros y devolviendo estructuras como valores de funciones.

No está permitido realizar comparaciones de estructuras.

Muchos compiladores que admiten la devolución de estructuras lo hacen de manera no-reentrante, dejando el valor de la estructura devuelta en una variable estática en lugar de en el stack.

Las estructuras automáticas no pueden inicializarse a pesar de que inicializar variables automáticas es equivalente a realizar una asignación. En el S.O. Unix se obtendría el mensaje de error "No auto aggregate inicialitation" como respuesta a este intento.

## Punteros a estructuras

Debido a las restricciones impuestas a las variables de tipo estructura normalmente se suelen utilizar punteros a estructuras para realizar el traspaso de estructuras como argumentos de funciones:

Incluso si se permiten asignaciones de estructuras, pasar una estructura como argumento puede no ser muy buena idea en ocasiones; por ejemplo:

```
#define LINEAS 24
#define COLUMNAS 80

struct ventana { /* una parte de una pantalla */
        char lineas [LINEAS][COLUMNAS];
        int ult_col;    /* esquina superior */
        int ult_fila;   /* esquina superior */
        int nlineas;    /* número de líneas */
        int ncols;      /* número de columnas */
} pantalla;

inicializa()
{
        pantalla.ult_col = 0;
        pantalla.ult_fila = 0;
        pantalla.nlineas = LINEAS;
        pantalla.ncols = COLUMNAS;

        borra_pantalla(pantalla.lineas);
        muestra(pantalla);
}
```

La llamada a "muestra" (definida en otro lugar) copia la totalidad de la estructura en el stack. Esta estructura contiene LINEAS * COLUMNAS = 24 × 80 = 1920 caracteres, así como algunas variables enteras. Compárese esto frente a la alternativa de pasar como argumento un puntero a la estructura, que requeriría tan sólo una variable entera (la dirección de la estructura).

Hay que destacar que pantalla.lineas es un array, y aquí se pasaría un puntero al primer elemento de este array en la llamada a la función borra_pantalla.

Los punteros a estructuras se declaran de la manera habitual, como los restantes punteros:

```
/* extrae elementos de la estructura
 * persona y los imprime
 */
ppersona(p)
struct persona *p;
{
        printf("Nombre: %s\n", p->nombre);
        printf("Dirección: %s\n", p->direccion);
        printf("Código Postal: %d\n", p->cod_postal)
        printf("Código de la S.S.: %d\n", p->ss_num)
        printf("Sueldo: %.2f\n", p->sueldo);
        printf("Fecha de nacimiento: %d %d %d\n",
                p->nacim_fecha.dia,
                p->nacim_fecha.mes,
                p->nacim_fecha.año);
```



```
        printf("Fecha de incorporación: %d %d %d\n",
                p->incorp_fecha.dia,
                p->incorp_fecha.mes,
                p->incorp_fecha.año);
```

La construcción

```
puntero-a-estructura -> miembro
```

es equivalente a:

```
(*puntero-a-estructura) . miembro
```

El operador "->" consiste en un signo menos "-" seguido por el signo mayor ">".

Los paréntesis en "(*p).nombre" son necesarios porque el orden de evaluación del operador "." (miembro de una estructura) es mayor que el del operador "*" (valor apuntado por). Así *emp.nombre es equivalente a emp.nombre[0].

Los operadores "." y "->" tienen mayor prioridad que los operadores aritméticos. Así pues, dada la declaración:

```
struct {
        int x;
        int y;
} *p;
```

la expresión

```
++p->x
```

incrementaría el valor del miembro "x" de la estructura apuntada por "p", en lugar de incrementar el puntero "p".

Análogamente:

```
*p->y;          obtiene el valor de 'y' apuntado por 'p'
*p->y++         idem., incrementando p->y
(*p->y)++       incrementa la 'y' apuntada por 'p'
*p++->y;        incrementa 'p' tras buscar p->y
```

Del mismo modo que C permite arrays de arrays como un tipo más de variables, también permite la definición de arrays de estructuras.

## Campos

Dentro de la declaración de una estructura puede aparecer un miembro seguido por un carácter de dos puntos ":" y una expresión constante, denominándose entonces campo **(field)**.

El tipo del miembro debe ser alguno de los tipos enteros (char, int, short, long o unsigned); la expresión constante especifica cuántos bits van a utilizarse para ese miembro.

Los campos de bit son una alternativa a las operaciones explícitas de desplazamiento y máscaras de bits. Consideremos el programa:

```
#define KEYWORD 01
#define EXTERNAL 02
#define STATIC 04

int flags;

...
        flags |= EXTERNAL | STATIC;
        ...
        if (flags & KEYWORD) {
                ...
```

Esto mismo se podría también haber escrito como:

```
struct {
        int is_keyword:1;
        int is_extern:1;
        int is_static:1;
} flags;

        ...
        flags.is_extern = flags.is_static = 1;
        ...
        if (flags.is_keyword) {
                ...
```

Desafortunadamente, el empleo de campos de bit en C tiene algunos inconvenientes:

- en las versiones antiguas de compiladores de C, todos los campos tienen que ser cantidades sin signo;
- no pueden emplearse datos binarios empaquetados de una máquina a otra, pues el empaquetado es dependiente de la máquina al no especificarse si la asignación de bits se hace de izquierda a derecha, o de derecha a izquierda;
- los campos de bit no son exactamente variables, en el sentido de que no pueden definirse punteros a campos de bit, ni arrays de campos, aunque, por supuesto, sigue siendo válido definir punteros a estructuras que contengan campos de bit.



## Uniones

Una unión consiste en una variable que puede contener valores de diferentes tipos, en lugar de un solo tipo como las variables "normales". Las uniones son una de las alternativas para solventar la falta de comprobaciones de tipo del lenguaje C en tiempo de ejecución.

La sintaxis de la declaración de una unión es semejante a la declaración de una estructura sin más que cambiar la palabra estructura por unión.

```
struct {        /* entrada en la tabla de simbolos */
        char *name;
        int type;
        union {
                int u_ival;
                float u_fval;
                char *_sval;
        } uval;
} symtab[NSYM];
```

Las variables symtab[i].uval tienen capacidad suficiente para contener variables de cualquiera de los tipos int, float o char, siendo responsabilidad del programador tener el cuidado necesario para controlar los tipos adecuados de las variables contenidas en una unión.

La unión definida anteriormente permitiría el acceso a sus distintos elementos como:

```
switch (symtab[i].type) {
case INT:
        printf("%d\n", symtab[i].uval.u_ival);
        break;
case STRING:
        printf("%s\n", symtab[i].uval_sval);
        break;
case FLOAT:
        printf("%f\n", symtab[i].uval_fval);
        break;
}
```

siendo similar al programa en Pascal:

```
type symtab =
    record
        name : array[1..10] of char;
        case stype : integer of
                int : ( ival : integer );
                float : (fval : real );
                string : (sval : array[1..20] of char );
    end;
```

```
var
   symtab : array[1..nsymbols] of symtab;
   ...
   case symtab[i].stype of
        int : writeln(symtab[i].ival);
        float : writeln(symtab[i].fval);
        string : writeln(symtab[i].sval);
   end
```

La enorme similitud entre ambos programas puede aún aumentarse mediante el empleo del preprocesador de C, empleando #defines que hagan más manejables los nombres de variables:

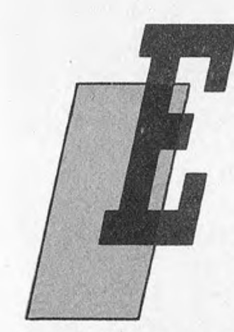
```
#define ival uval.u_ival
#define fval uval.u_fval
#define sval uval.u_sval
```

Las operaciones que pueden realizarse sobre las uniones son las mismas que las permitidas sobre las estructuras: acceder a uno de sus miembros y acceder a la dirección de la unión.



# CAPITULO X

## ENTRADA/SALIDA EN LENGUAJE C



El lenguaje C no incorpora directamente instrucciones para la realización de las operaciones de Entrada/Salida, implementándola a través de librerías y funciones.

Existe un conjunto de funciones de E/S que es portable a un gran número de sistemas operativos (UNIX, VMS, MS-DOS, CP/M, ...), implementadas en la denominada librería de E/S estándar. Cualquier programador de C debe conocer los nombres y argumentos de estas funciones, considerando que van a ser las mismas en cualquiera de los sistemas operativos mencionados.

Además se incluyen funciones para permitir el interface directo con las propias funciones de E/S del sistema operativo.

### Los ficheros en la librería estándar de E/S

Las declaraciones necesarias para el manejo de la librería estándar de E/S son accesibles mediante el empleo de un #include:

```
#include <stdio.h>
```

Este fichero contiene las declaraciones de variables y definiciones de macros necesarias.

En algunos sistemas puede llegar a ser necesario notificar expresamente al compilador la inclusión de la librería estándar. Por ejemplo, en los antiguos sistemas Unix Versión 6 se requería la in-

clusión de la opción "-1S" al compilar. Esto ya no es necesario en los sistemas Unix Versión 7 actuales o en sus derivados.
Internamente, la librería estándar de E/S almacena la información correspondiente a cada fichero abierto en una estructura, identificando a los ficheros por medio de punteros a estructuras. El fichero <stdio.h> define FILE como una de estas estructuras.

## Apertura de ficheros

Antes de que se pueda acceder a un fichero éste debe ser abierto mediante la función fopen:

```
FILE *fopen(name, mode) /* Abre un fichero */
char *name;             /* Nombre del fichero a abrir */
char *mode;             /* Modo de acceso al fichero */
```

El primer argumento de fopen es el nombre del fichero, enviado como una cadena de caracteres. El formato del nombre de fichero es dependiente del sistema operativo. Sin embargo, muchas (no todas) las implementaciones de la librería C estándar de E/S trasladan los nombres de ficheros especificados como en Unix al formato requerido por el sistema operativo.
Por ejemplo, el formato de un nombre de fichero en Unix es:

/directorio1/directorio2/ ... /fichero

Cada uno de los componentes del nombre del fichero (directorio1, directorio2, fichero) pueden estar formados por cualquier secuencia de caracteres excepto por un slash "/". Sólo son significativos los catorce primeros caracteres. No hay un límite para el número de directorios que se pueden especificar.
El "mode" (modo) puede ser de tres tipos:

- modo de acceso "r" indica que el fichero se va a abrir para lectura. Si el fichero no existe, se devolverá un error.
- modo "w" especifica que el fichero se va a abrir para escritura. Si el fichero ya existe no se tendrá en consideración el contenido anterior, perdiéndose por completo. Si el fichero no existiese, entonces lo creará la propia función fopen.
- modo "a" especifica que el fichero se va a abrir para añadirle texto (append). Si el fichero ya existe, los nuevos datos se grabarán al final del mismo, y si no existiese será creado por la función.

En caso de que se produzca un error, fopen devuelve el valor NULL, definido en <stdio.h> como (char *)0.



## E/S básica

Las operaciones más sencillas que se pueden realizar sobre un fichero que ya esté abierto son getc y putc.

```
int getc(fp)
FILE *fp;
```

getc devuelve el siguiente carácter del fichero especificado for fp, o EOF (definida en <stdio.h>). EOF se devuelve cuando se produce algún error o se alcanza el final de fichero.
Análogamente:

```
int putc(fp)
FILE *fp;
```

putc escribe el carácter dado en el fichero fp, y retorna "c" o EOF (en caso de error).

```
feof(fp)
FILE *fp;
```

feof devuelve un valor distinto de cero si se ha llegado al final del fichero fp.

```
ferror(fp)
FILE *fp;
```

ferror devuelve un valor distinto de cero si se ha encontrado algún error durante la lectura o escritura del fichero fp.

```
fclose(fp)
FILE *fp;
```

fclose cierra el fichero fp, devolviendo un EOF si se produce algún error (por ejemplo, si el fichero fp no fue abierto previamente).
La función freopen

```
FILE *freopen(name, mode, fp)
char *name;
char *mode;
FILE *fp;
```

en primer lugar cierra el fichero especificado por fp y abre un nuevo fichero, de modo que el fichero recién abierto reutiliza la es-

tructura FILE apuntada por fp. Freopen devuelve fp si la apertura se ha realizado sin problemas, o NULL en caso contrario.

```
ungetc(c, fp)
char c;
FILE *fp;
```

ungetc provoca que la siguiente llamada a getc(fd) (así como scanf, getw, gets, y las restantes funciones de lectura) lea el carácter "c". Mediante el empleo de ungetc sólo se puede poner un carácter en el buffer del fichero fd.

## Ficheros estándar

Cuando se inicia la ejecución de un programa C se produce la apertura automática de tres ficheros, antes de que llegue a producirse la llamada a main(). Estos ficheros son la entrada estándar "stdin" (standard input, normalmente el teclado), la salida estándar "stdout" (standard output, normalmente la pantalla de vídeo) y el fichero estándar de salida de errores "stderr" (standard error, normalmente la pantalla de vídeo).

Este encaminamiento normal de los ficheros estándar (pantalla y teclado por defecto), puede verse modificado a través del intérprete de comandos de Unix y reencaminado hacia ficheros, "tuberías" (pipes) o regiones de memoria compartida.

La librería estándar de E/S declara stdin, stdout y stderr como punteros a estructuras FILE para los ficheros de entrada estándar, salida estándar y fichero estándar de errores, respectivamente.

Las funciones getchar y putchar son macros, en lugar de auténticas funciones, definidas en <stdio.h> como:

```
#define getchar() getc(stdin)
#define putchar(c) putc(c, stdout)
```

Hay que advertir que stdin, stdout y stderr son constantes y no pueden ser reasignados mediante fopen como podría parecer natural.

```
stdout = fopen("myfile", "w");
```

no es un procedimiento válido para redireccionar la salida estándar al fichero myfile. Sin embargo,

```
freopen("myfile", "w", stdout);
```



sí podría servir para este propósito, debido a que freopen cierra y abre el fichero, como comentamos anteriormente.

## Salida con formato

printf es la función básica utilizada para salida con formato; es llamada en la forma:

```
printf(control, arg1, arg2, ...)
char *control;
```

Esta función acepta un número variable de argumentos, convirtiéndolos, formateándolos e imprimiéndolos bajo las especificaciones incluidas en "control". Los caracteres del string de control se vuelcan en el fichero de salida estándar, salvo el metacarácter "%", que se emplea como una especificación de formato para controlar la impresión de los argumentos.

Las especificaciones de formato de salida comienzan con un "%" y terminan con un *carácter de conversión*. Una especificación puede contener opcionalmente, descritos por orden:

- un signo menos "-", indicando que el siguiente argumento debe imprimirse justificado a la izquierda en su campo de salida;
- una cadena de dígitos, especificando el tamaño mínimo del campo de salida para ese argumento. Si el argumento ya formateado ocupa menos caracteres que la anchura del campo que le ha sido asignado se completará con caracteres por la izquierda (o por la derecha si se ha especificado justificación a la izquierda).

Si la especificación de la anchura del campo contiene un cero como primer carácter (por ejemplo 05d), la longitud total del campo se completará con ceros en lugar de con espacios en blanco. La inclusión de este cero por la izquierda no debe confundirse en ningún caso con la expresión de una constante en octal (por ejemplo, el carácter espacio en blanco, ASCII 32 en decimal, se expresa como 040 en octal).

A diferencia de la salida con formato en Fortran, printf no trunca la conversión de argumentos en caso de que necesiten una extensión mayor que la especificada para el campo de salida;

- un punto, separando el ancho del campo de salida, de la precisión;

- una cadena de dígitos que puede especificar tanto el número máximo de caracteres a imprimir de una cadena, como el número de dígitos a imprimir a la derecha del punto decimal cuando se están imprimiendo números en coma flotante (float o double);
- una ele "l", que indica que el siguiente argumento es un long it, en lugar de un int;
- algunas implementaciones de printf interpretan el carácter asterisco "*" como una anchura de campo o precisión, con el significado de que el siguiente argumento indicará realmente la anchura del campo o precisión del argumento a imprimir. Por ejemplo:

  ```
  printf("%.*s", longitud, cadena);
  ```

  imprimirá al menos "longitud" caracteres de la cadena de caracteres "cadena".

Los caracteres de conversión de salida son:

d - Convierte el argumento entero a un número decimal con signo.
o - Convierte su argumento (entero) a base octal.
x - El argumento entero se convierte a hexadecimal.
u - El argumento entero se convierte a notación decimal sin signo.
c - Interpreta su argumento como un único carácter.
s - Considera su argumento una cadena (string).
e - Convierte su argumento en coma flotante (interpretado como double, porque C siempre pasa los números en coma flotante como double) a notación exponencial, con signo, mantisa y exponente:

  [-]m.nnnnE[-]xx

  La longitud de la cadena de enes "n" viene determinada por la precisión, que es de 6 por defecto. (Recordemos que float proporcionaba 6 dígitos de precisión y double 15.)

f - El argumento en coma flotante se expresa en notación de coma fija:

  [-]mmm.nnnn

  La longitud de la cadena de enes "n" se toma nuevamente de 6 por defecto.



g - Convierte su argumento en coma flotante a formato "%f" o "%e", dependiendo de cuál de los dos proporcione una cadena de menor longitud.

Cualquier otro carácter distinto de los especificados anteriormente y que siga a un signo "%" se imprimirá literalmente. Así:

```
printf("%%");
```

Imprimirá un único carácter "%".

La aparición de los anteriores caracteres en mayúsculas (D, O, X, U, E, F o G) se interpreta como si una ele "l" precediese al carácter correspondiente en minúculas. Así

```
printf("%D", valor);
```

es equivalente a

```
printf("%ld", valor);
```

La salida con formato especificada por printf y enviada por esta función al fichero de salida estándar también se puede enviar a otro fichero o a una cadena de caracteres, por medio de las funciones fprintf y sprintf, respectivamente.

```
fprintf(fp, control, arg1, arg2, ...)
FILE *fp;
char *cntrol;
```

realiza su salida sobre el fichero especificado por fp.

```
sprintf(string, control, arg1, arg2, ...)
```

lleva su salida al array string de acuerdo con las especificaciones de formato incluidas, en lugar de poner la salida en un fichero.

## *Entrada con formato*

scanf es la función básica de entrada con formato.

```
int scanf(control, arg1, arg2, ...)
char *control;
```

Esta función lee caracteres con formato a partir del fichero de

entrada estándar "stdin", interpretándolos de acuerdo a las especificaciones de la cadena de control.

Cada uno de sus argumentos debe ser un puntero, especificando la dirección de la variable donde deberá almacenarse el resultado. Obviamente, scanf es una función que necesita "llamada por referencia", pues va a modificar las variables que se le pasan como argumentos. Un error frecuente en su uso consiste en la omisión de los "&" en cada uno de los argumentos.

Los caracteres de separación como espacios en blanco, tabuladores y caracteres de salto de línea que figuren en la cadena de control, serán ignorados.

Los restantes caracteres (excepto el "%") se corresponden con una especificación de formato de entrada, pudiendo consistir en:

- el carácter asterisco "*", indicando que el valor leído no debe almacenarse en la variable;
- una cadena de dígitos especificando la longitud máxima del campo de entrada (el número de caracteres a convertir);
- carácter "h", para indicar que el siguiente argumento debe ser considerado como un puntero a un short int, o el carácter "l" indicando que debe ser tratado como un puntero a un long int, o a un double (equivalente a long float);
- carácter de conversión.

Las conversiones de caracteres de entrada permitidas son:

d - Lee un entero decimal. El argumento debe ser un puntero a un entero (o un puntero a un short int o long int, si se hubiesen antepuesto "h" o "l" a la especificación de formato "d").
o - Lee un número entero octal.
x - Lee un número entero hexadecimal.
c - Lee un único carácter de la entrada. El argumento debe ser un puntero a un carácter.
s - Acepta una cadena de caracteres terminada por un espacio en blanco o un carácter de salto de línea. El array de caracteres donde se guarde el resultado debe ser lo bastante grande como para contener todos los caracteres y el carácter nulo terminador de cadena.
e - Lee un número en coma flotante. El argumento debe ser un float (o un double, con especificación "le").
f - Análogo al anterior. Acepta un número en coma flotante.
[ - Lee una cadena de caracteres limitada por separadores arbitrarios, almacenándola en el array apuntado por el argumento. El corchete de apertura "[" va seguido por un conjunto de caracteres y un corchete de cierre "]". Si el carácter inmediatamente posterior a "[" no es un circunflejo "ô", entonces scanf leerá solamente los caracteres que aparezcan en el conjunto de caracteres entre corchetes. Por el contrario, si el carácter inmediatamente posterior a "[" es un circunflejo "^", entonces scanf leerá caracteres hasta que encuentre algún carácter de los contenidos en el conjunto de caracteres entre corchetes.



Los caracteres de conversión d, o, x, e, y f pueden aparecer en mayúsculas o precedidos por una ele "l", para indicar que el correspondiente puntero apunta a un long int o long float.

scanf interrumpe la lectura cuando llega al final de la cadena de control, o cuando la entrada no cumple la especificación de formato impuesta (por ejemplo, intentar leer un literal con una especificación %d de número entero).

scanf devuelve un número entero indicando el número de asignaciones que ha realizado o EOF si ha encontrado un final de fichero y esperaba leer más caracteres.

Es importante aclarar que el valor cero no quiere decir que se haya alcanzado el final de fichero, sino que no se ha convertido correctamente ningún campo de entrada.

Análogamente a lo descrito para printf, scanf también permite que su entrada provenga de una cadena de caracteres o de un fichero.

```
int fscanf(fd, control, arg1, arg2, ...)
FILE *fd;
char *control;
```

es similar a scanf, con la diferencia de que toma su entrada del fichero fd, en lugar de "stdin".

```
int sscanf(control, arg1, arg2, ...)
char *control;
```

tomaría su entrada de una cadena de caracteres. Como la lectura de una cadena no viene acompañada de efectos laterales (side effects), sscanf es generalmente más útil que scanf o fscanf.

## E/S orientada a líneas

La función fgets leerá la siguiente línea de entrada (incluido el carácter \n) del fichero "fp", guardando su entrada en el array

"line" que deberá estar terminado en un carácter nulo. Se leerán al menos "n-1" caracteres. fgets devuelve line, salvo que se haya alcanzado el final de fichero antes de llegar a leer ningún carácter.

```
char *fgets(line, nchars, fp)
char *line;
int nchars;
FILE *fp;
```

La función fputs escribe la cadena de caracteres "line" sobre el fichero especificado por fp.

```
fputs(line, fp)
char *line;
FILE *fp;
```

La librería estándar también incluye las funciones gets y puts que leen y escriben de stdin y stdout, respectivamente. Sin embargo, no son equivalentes a fgets y fputs dirigidas a stdin y stdout.

La función gets

```
char *gets(line)
char *line;
```

lee la siguiente línea de entrada en stdin (incluyendo el \n) sobre el array de caracteres "line", que se supone tendrá suficiente espacio para contener la entrada. El carácter \n no se almacena en "line". A diferencia de fgets, gets devuelve "line" a menos que se alcance el final de fichero antes de que se haya podido leer ningún carácter.

La función puts

```
puts(line)
char *line;
```

escribe la cadena de caracteres "line" sobre stdout seguido por un carácter \n.

## Entrada/Salida binaria

Las funciones básicas de E/S getc y putc son capaces de leer y escribir cualquier tipo de fichero (al menos bajo Unix), no sólo ficheros de texto.

Se puede escribir una "palabra" (tipo de dato entero) sobre un fichero utilizando:



```
int putw(word, fp)
int word;
FILE *fp;
```

putw devuelve la palabra escrita en el fichero, o EOF, si se ha producido algún error.

Se puede leer una palabra de un fichero utilizando

```
int getw(fp)
FILE *fp;
```

getw devuelve la palabra, o EOF, si se ha alcanzado el final de fichero antes de leer la totalidad de la palabra. Como EOF es un valor permitido para un int (pero no para un char), debería utilizarse feof para comprobar la condición de final de fichero después de getw.

Cualquier variable de C puede escribirse como si se tratase de un array de caracteres. Por ejemplo, dada la declaración:

```
union {
        long int val;
        char c[sizeof long];
} x;
```

Leyendo o escribiendo el array de caracteres x.c se transferirá exactamente el valor de x.val.

La función general de lectura binaria es fread:

```
int fwrite(pointer, size, number, fp)
char *pointer;
unsigned int size;
unsigned int number;
FILE *fp;
```

Esta función lee un número "number" de objetos del fichero "fp", de un tamaño de "size" bytes cada uno de ellos, en el área de datos apuntada por "pointer". fread devuelve el número de objetos completos que ha podido leer. Así, cero equivale a un fin de fichero.

La función general de escritura binaria es fwrite

```
int fread(pointer, size, number, fp)
char *pointer;
unsigned int size;
unsigned int number;
FILE *fp;
```

que escribe "number" objetos del fichero "fp", de "size" bytes cada uno de ellos, en el área apuntada por "pointer". fwrite devuelve el

número de objetos completos escritos, que coincidirá con number a menos que se haya producido algún error.

Un ejemplo típico de utilización sería:

```
long int x;
FILE *from, *to;
...
while (fread((char *)&x, sizeof x, 1, from) == 1)
        fwrite((char *)&x, sizeof x, 1, to);
```

copiando un long del fichero "from" al fichero "to".

Es importante observar el empleo del cast &x para obtener un char *. La expresión &(char *)x que fuerza a que "x" sea del tipo char, no funcionaría porque (char)x no es una variable, y no se puede aplicar un &.

Aunque la utilización de fread y fwrite es portable, los ficheros manejados por ellos no tienen porqué serlo, pues estos ficheros son fiel reflejo del tamaño y formato de los objetos de datos representados por el ordenador en cuestión.

Si se necesita escribir ficheros portables es mejor emplear funciones de desplazamiento y máscaras de bits o utilizar datos con formato.

## Acceso aleatorio a ficheros

La E/S de ficheros en lenguaje C es habitualmente secuencial. Sin embargo, un fichero puede ser leído o escrito en cualquier orden.

La función fseek

```
fseek(fp, offset, origin)
FILE *fp;
long int offset;
int origin;
```

obliga a que la siguiente llamada a getc o putc tenga lugar sobre la posición offset del fichero. El origen es 0, 1 ó 2 indicando si el offset es relativo al principio del fichero (0), a la posición actual (1) o al final del fichero (2).

La función

```
long int ftell(fp)
FILE *fp;
```

devuelve la posición actual del offset a partir del principio o del final del fichero fp.



La función

```
rewind(fp)
FILE *fp
```

es equivalente a:

```
fseek(fp, 0L, 0);
```

## Entrada/Salida con buffer

Las llamadas al núcleo del sistema (system calls) para realizar operaciones de E/S ocupan una gran cantidad de tiempo, comparadas con una llamada a función. Una llamada al núcleo consiste en utilizar funciones del núcleo del sistema operativo de una forma semejante a como se manejan las funciones "normales" en un programa.

Para realizar una llamada al núcleo del sistema se requiere un trabajo extra, comparándolo con una llamada a función, necesitándose almacenar información adicional durante la llamada y el retorno, además de la sobrecarga necesaria para encontrar un manejador de la llamada al núcleo, traspasar argumentos, etc.

La sobrecarga en las llamadas al núcleo se puede disminuir mediante la "bufferización": se puede transferir un único carácter de/hacia un buffer desde el proceso de usuario; una vez que el buffer se ha llenado (para ficheros de salida) o vaciado (para ficheros de entrada) se puede transferir la totalidad del buffer lleno de caracteres.

Este procedimiento reduce el tiempo de ejecución de un programa, pero puede tener otros efectos. En particular, si putchar pone caracteres en un buffer, estos caracteres no aparecerán en la pantalla a medida que vayan siendo enviados por putchar; es más, ni siquiera lo harán después de que se haya producido el retorno de putchar, con la molestia de que no aparece eco inmediato en pantalla.

Si un programa termina anormalmente por cualquier causa, la salida que tenía bufferizada se quedará sin enviar al fichero de salida.

Del mismo modo, los errores que se estén produciendo en la escritura no aparecerán hasta que se hayan escrito los datos del buffer, lo cual podría suceder bastante después de que la llamada a putc haya enviado los datos.

La "bufferización" de entrada causa generalmente pocos problemas, salvo en el caso particular de que más de un proceso esté leyendo simultáneamente el mismo fichero.

Realmente no hace demasiada falta controlar la bufferización de los datos; dejándolo en manos de la librería estándar de E/S podemos confiar en que la librería "hará las cosas bien" sin necesidad de control (por ejemplo no bufferizará los datos de salida a un terminal).

No obstante, la librería estándar de E/S posee funciones que permiten validar o invalidar la "bufferización" sobre un fichero, así como una función que fuerza a la librería de E/S a que envíe de inmediato los datos retenidos en el buffer.

La función setbuf

```
setbuf(fp, buffer)
FILE *fp;
char *buffer;
```

informa a la librería estándar de E/S de que utilice el array buffer de BUFSIZ caracteres (definido en la <stdio.h>) para bufferización. Si buffer es igual a NULL entonces el fichero no estará bufferizado. setbuf debería llamarse antes de que se vaya a realizar cualquier operación de E/S sobre el fichero.

La función

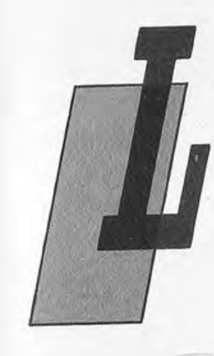
```
fflush(fp)
FILE *fp;
```

fuerza que cualquier dato bufferizado del fichero fp sea escrito en el fichero. fflush devuelve EOF para indicar un error (por ejemplo, la imposibilidad de grabar los datos en el fichero) o cero en el caso contrario.



# CAPITULO XI

## EFICIENCIA Y PORTABILIDAD EN EL C



La capacidad de escribir programas que utilicen con eficiencia los recursos del sistema, estén libres de errores (error-free) y sean fácilmente transportables a otros computadores son los signos que indican el paso de un buen programador y el uso de un lenguaje adecuado.

### Eficiencia

Cuando hablamos de eficiencia en un programa de ordenador nos referimos tanto a su velocidad de ejecución como a la utilización óptima de los recursos críticos del sistema (memoria y ocupación en disco, básicamente). No basta con que los programas funcionen, sino que además deben hacerlo de manera que supongan la menor carga posible para el ordenador y no gasten sus recursos inútilmente.

Normalmente, la optimización de un aspecto del programa suele degradar otros. Por ejemplo, hacer que un programa se ejecute más rápido normalmente requiere que se duplique el código fuente en algunos casos, en lugar de realizar llamadas a funciones. También se puede conseguir un ahorro de ocupación en disco, utilizando empaquetado de datos, a costa de degradar la velocidad de acceso a los datos. Estas y otras posibilidades de mejora de la eficiencia pueden ser frustrantes para los no-programadores, a quienes les puede costar comprender cómo la mejora de una característica de un programa puede afectar negativamente a otras, llegando incluso a empeorar el resultado final.

Afortunadamente, existen algunas técnicas de programación que mejoran siempre la eficiencia en cualquier sistema que consideremos.

Por ejemplo, el uso de operadores de incremento ++ y decremento -- es más rápido y compacto que la utilización de operaciones como x = x + 1 (frente a x++).

La utilización de funciones puede resultar perjudicial en ocasiones, como es en el caso de una función que sea llamada desde el interior de un bucle un gran número de veces. Pensemos en el siguiente ejemplo:

```
/* Función que evalúa el valor del polinomio
 * x + 3x -1 en el intervalo de valores de 0 a 100
 */

for (i = 0; i < 100; i++) {
        x = polin((double) i);
        printf(" d  fn", i, x);
}


double polin(z)
double z;
{
        return ((z + 3) * z -1);
}
```

Sería más eficiente la escritura del programa en la forma:

```
for (i = 0; i < 100; i++) {
        x = (double) i;
        x = ((x + 3) * x -1);
        printf (" d  fn", i, x);
}
```

donde hemos eliminado la llamada a la función polin, que debería realizarse 100 veces, con lo que esto significa en cuanto a secuencias de llamada y retorno de función y manejo de stack: la llamada requiere salvar el estado de determinados registros de la CPU mediante la instrucción push de ensamblador, y el retorno requiere la recuperación de sus valores originales mediante la instrucción pop; en ocasiones la función exige además la utilización de variables automáticas que han de ser definidas en cada llamada a función.

La utilización de punteros en lugar de índices de arrays genera un código más compacto en todos los casos, si tenemos en cuenta que internamente el compilador convierte todas las referencias a subíndices en manejo de punteros. Por ejemplo,

```
for (j = 0;;) {
        i += a[j++];
        ...
}
```



accedería al elemento "j" del array creando internamente un puntero al elemento 0 del array y desplazando el puntero en "j" elementos, incrementando posteriormente "j" en 1. Por el contrario:

```
char *p = a;
for (;;) {
        i = *(p++);
        ...
}
```

Sería más eficiente porque "p" ya es directamente un puntero al elemento "j" del array al que deseamos acceder. El manejo de punteros es muy utilizado en los casos en que se desea acceder secuencialmente a todos los elementos de un array.

En algunos círculos C es considerado como un lenguaje críptico, difícil de leer y de escribir. Esta mala reputación es debida íntegramente al estilo de algunos programadores, amigos de escribir programas difíciles de comprender incluso para ellos mismos cuando ha transcurrido algún tiempo desde su escritura, debido a la utilización de nombres de variables inadecuados, problemas muy complejos llenos de "trucos" y utilización de expresiones complicadas en la misma línea sin hacer uso de variables intermedias. Todo esto puede hacer en ocasiones que los programas sean un poco más rápidos, al precio de hacerlos ininteligibles, siendo más razonable plantearse de antemano la idea de si no sería posible mejorar el algoritmo empleado, reducir el número de llamadas a funciones o, en casos muy extremos, codificar alguna subrutina en lenguaje ensamblador, aunque esto requiere una muy buena razón para hacerlo.

## Portabilidad

La portabilidad en lenguaje C se refiere a la posibilidad de transportar programas de uno a otro ordenador y ponerlos en funcionamiento con sólo volver a compilarlos. La portabilidad se refiere sólo a los programas fuente; no tiene sentido alguno intentar transportar programas compilados de uno a otro ordenador, pues han sido compilados haciendo uso de las facilidades suministradas por un determinado hardware, que no tiene porqué estar presente sobre otro, salvo que expresamente nos estemos refiriendo a ordenadores compatibles y funcionando bajo un mismo sistema operativo.

Posibles causas de pérdida de portabilidad son la utilización de "números mágicos" o constantes dependientes del ordenador

considerado, como pueden ser el tamaño en bytes de los buffers del sistema, la utilización de secuencias de escape y control particulares para determinados periféricos y el empleo de funciones específicas de un sistema operativo que no tienen por qué estar presentes o funcionar igual en otros sistemas.

Para evitar el empleo indiscriminado de "números mágicos" es conveniente el empleo de las instrucciones #define e #ifdef del preprocesador de lenguaje C, de manera que permita la modificación de determinadas constantes antes de proceder a la compilación del programa. Por ejemplo:

```
#define UNIX 1
#define GCOS 2
#define MS_DOS 3

"..."
flag = UNIX;
"..."

#if flag == UNIX
        "..."
        Parámetros propios del S.O. UNIX
        "..."
#endif

#if flag == MS_DOS
        "..."
        Parámetros propios del S.O. MS_DOS
        "..."
#endif
```

De este modo podríamos emplear parámetros y funciones propios de cada sistema operativo con la seguridad de que los programas serían independientes del sistema considerado.

Una causa posible (y peligrosa) de pérdida de portabilidad es la redefinición de nombres de funciones, como los contenidos en la librería de C estándar. Si redefinimos funciones como printf, strcpy, strlen, etc., por otras funciones propias que teniendo el mismo nombre manejen distintos argumentos de llamada y retorno, tendríamos problemas de funcionamiento obvios.

Otra causa de pérdida de portabilidad es la utilización de secuencias de escape y control específicas para determinados periféricos, como pantallas e impresoras de una determinada marca. Si nuestro programa ha de funcionar algún día con otros periféricos tendremos serios problemas de funcionamiento. La adopción de las secuencias de escape y control apropiadas para cada periférico ha de efectuarse en tiempo de ejecución del programa, en lugar de en tiempo de compilación. Lo más normal es proceder a la lectura de un fichero de parámetros conteniendo las características de cada periférico considerado. Otra posibilidad (mejor) es la utilización de funciones de librería que hagan uso de



esta posibilidad (lectura de un fichero de parámetros) y además utilicen algún tipo de optimización, como el empleo de determinadas funciones que no están presentes en todos los terminales (borrado e inserción de líneas, por ejemplo), o manejen por software otras posibilidades, como ventanas de texto. Estas posibilidades y otras son manejadas por un paquete estándar como es el curses de la Universidad de Berkeley, existente en la mayoría de las instalaciones que trabajan con lenguaje C.

# CAPITULO XII

## *UN EJEMPLO COMPLETO: EL PROGRAMA 'calles.c'*




modo de glosario de lo visto hasta ahora vamos a desarrollar un programa que maneje muchos aspectos interesantes del lenguaje C, presentes en la mayoría de los programas que vayamos a realizar.

El programa 'calles.c' tiene como misión modificar el nombre de una calle (que se pasará como argumento de entrada) realizando sobre él una serie de abreviaturas (por ejemplo, cambiar 'teniente' por 'tte' o 'señor' por 'sr') y eliminando palabras no deseadas ('de', 'el', 'la', ...). Se podría utilizar para minimizar el efecto de posibles duplicaciones en una base de datos que contuviese nombres de calles, con entradas como:

"SANTA CATALINA, PARQUE DE"
"SAN ANDRES, CALLE"

El programa devolvería como salida los nombres:

"STA CATALINA PQE"
"S ANDRES CALLE"

donde ha reducido las palabras 'SANTA' y 'SAN' cambiándolas por 'STA' y 'S', y eliminado la partícula 'DE' y las comas de separación.

Pensemos ahora en una gran base de datos, con 50.000 o más nombres de calles, que constituyan un índice, y el efecto de entradas como:

"SAN ANDRES, CALLE"

"S. ANDRES CALLE DE"
"DE SAN ANDRES, CALLE"
"SAN ANDRES CALLE"

Nuestro programa evitaría esta dispersión de nombres y los transformaría en el único nombre:

"S ANDRES CALLE"

facilitando su localización y eliminando la posibilidad de claves duplicadas por error.

El ejemplo no es muy importante en sí mismo, pero maneja conceptos aislados que sí son de interés; argumentos en la línea de llamada a un programa, manejo de arrays y estructuras con punteros, funciones propias y de la librería C estándar, y estructuras de control muy empleadas en C (for, if, if else, switch).

Para compilar el programa 'calles.c' habremos de utilizar el comando de sistema operativo:

cc -o calles calles.c

y ejecutar el programa como:

calles 'SAN ANDRES, CALLE DE'

obteniendo como resultado:

Entrada: ->SAN ANDRES, CALLE DE <-
Salida: ->S ANDRES CALLE<-

El listado del programa, casi sin ningún tipo de comentario para facilitar su seguimiento y el de su estructura, es el que sigue (al final del capítulo se incluye con todos los comentarios autoexplicativos):

```
#include <stdio.h>

char calle[30], palabra[30];
char * tabla();

struct mapa {
        char *car_in;
        char *car_out;
};

struct mapa calles[] = {
        "ALCALDE",          "ALC",
        "ALFEREZ",          "ALFZ",
        "ALMIRANTE",        "ALMTE",
        "ARQUITECTO,"       "ARQTO"
```



```
        "DE",               (char *)0,
        "EL",               (char *)0,
        "NUESTRA",          "NRA",
        "PARQUE",           "PQE",
        "PRESIDENTE",       "PDTE",
        "PUERTO",           "PTO",
        "REPUBLICA",        "RPBCA",
        "SAN",              "S",
        "SANTA",            "STA",
        "SANTO",            "STO",
        "SEÑOR",            "SR",
        "SEÑORA",           "SRA",
        "TENIENTE",         "TTE",
        "VIRGEN",           "VG"
};

short hi_calle = sizeof(mapa) / sizeof(struct calles);

/* Programa principal */

main(argc, argv)
int argc;
char *argv[];
{
        unsigned i;
        char *p;

        for (i = 0, p = argv[1]; *p; p++) {
                switch (*p) {
                        case ' ':       /* separadores de */
                        case '.':       /* palabra        */
                        case ',':
                                palabra[i] = '\0';
                                strcat(calle, tabla(palabra));
                                strcat(calle, " ");
                                i = 0;
                                break;
                        default:
                                palabra[i++] = *p;
                }
        }
        palabra[i] = '\0';
        strcat(calle, tabla(palabra));
        strcat(calle, " ");
        trimar(calle);
        printf("\nEntrada: ->%s<-", argv[1]);
        printf("\nSalida : ->%s<-\n", calle);
}


/* Función tabla(s) */

char * tabla(s)
char s[];               /* argumento de llamada a 'tabla' */
{
        struct calles *p;
        int low, high, mid, cond;
        low = 0;
        high = hi_calle;

        while (low <= high) {
                mid = (low + high) / 2;
                p = &mapa[mid];
```

```
                if ((cond = strcmp(s, p->car_in)) < 0)
                        high = mid - 1;
                else if (cond > 0)
                        low = mid + 1;
                else
/* ¡Encontrado! */              return ((char *) p->car_out);
        }
        return ((char *) s);
}


/* Función trimar(s) */

void trimar(s)
char *s;
{
        char *p;
        unsigned dbl = 0;

        for (p = str; *p == ' '; p++)
                ;
        while (*p) {
                if (*p == ' ') {
                        dbl = 1;
                        p++;
                        continue;
                }
                if (dbl) {
                        *s++ = ' ';
                        dbl = 0;
                }
                *s++ = *p++;
        }
        *s = '\0';
}
```

Al principio del programa se declaran las variables globales calle[30] y palabra[30], como cadenas de caracteres, y se declara tabla como una función que devuelve un puntero a una cadena de caracteres.

Luego se define la estructura mapa, compuesta por car_in y car_out, punteros a las cadenas de caracteres de entrada y salida, respectivamente. calles se define como una estructura mapa, y se procede a inicializarla.

El número de elementos que contiene la estructura calles se calcula como

short hi_calle=sizeof(calles) / sizeof(struct mapa);

Es decir, el número de elementos (hi_calle) equivale al tamaño (sizeof) de la estructura calles, dividido por el tamaño de un elemento (mapa).

En este caso main aparece como main(argc, argv), debida a que va a manejar argumentos en la línea de llamada (el nombre



de la calle a modificar). El argumento se pasa entre comillas simples "'", para evitar que los espacios en blanco intercalados hagan que sea considerado como varios argumentos.

Mediante un bucle for y el desplazamiento de un puntero arg[1], se va utilizando el argumento de entrada carácter a carácter. Si se encuentra un carácter distinto del separador (los separadores son '', ':' y ','), se carga en el array "palabra" y se incrementa el subíndice "i".

Si, por el contrario, se alcanza un separador entonces se añade un carácter nulo a «palabra», como terminador de cadena de caracteres, y se envía como argumento a la función tabla, que nos devolverá un puntero a una cadena, bien apuntando a una cadena distinta de la de entrada, si la encuentra en la estructura calles, o la misma cadena de entrada si no la ha encontrado. Al final le añade un espacio en blanco como separador, y reinicializa el subíndice "i" a 0, para preparar la búsqueda de otra palabra.

Esta secuencia se repite hasta alcanzar el final de la cadena de entrada argv[1].

La función tabla tiene como misión efectuar la búsqueda de una palabra en la estructura calles, devolviendo la palabra modificada en caso de encontrarla. La búsqueda en la estructura se implementa mediante una búsqueda binaria.

Se compara la cadena a buscar con la correspondiente a la mitad de la tabla (que ha de estar ordenada), si es mayor la cadena buscada se repite la búsqueda en la mitad superior de la tabla, si no en la inferior; y así sucesivamente hasta encontrar el elemento buscado o finalizar con una comparación entre dos posibles elementos. Si al final del proceso no se ha encontrado la cadena buscada, se devuelve un puntero a la misma cadena, para que el programa principal la añada a la cadena obtenida como resultado de salida.

Al final, se llama a la función trimar para que elimine los espacios en blanco que pudiera haber al principio, final, y embebidos en la cadena obtenida como resultado del programa y se imprimen para su comparación las cadenas de entrada y salida.

La función trimar elimina los caracteres en blanco que pudiera haber en una cadena de caracteres, mediante el desplazamiento de un puntero. Cuando encuentra un blanco intermedio, pone a uno un indicador (dbl) y continúa la exploración de la cadena sin copiar ningún carácter a su salida. Cuando se encuentre un carácter distinto de blanco, si tiene a 1 el indicador dbl, entonces pone un blanco a su salida, y reinicializa dbl a 0. Luego copia sobre su salida el carácter distinto de blanco que acababa de encontrar. De este modo, ha convertido la aparición de múltiples blancos en uno solo. Al finalizar la exploración de la cadena, sitúa un carácter nulo como terminador de cadena de caracteres.

Para finalizar vamos a incluir el listado completo del programa, con todos sus comentarios autoexplicativos.

```
/***************************************************************
 * Define los arrays de caracteres 'calle' y 'palabra'        *
 * con capacidad para 30 caracteres, y 'tabla' como           *
 * una función que retorna un puntero a un char               *
 ***************************************************************
 */
char calle[30], palabra[30];
char * tabla();

/***************************************************************
 * Define una estructura 'mapa' compuesta por dos             *
 * punteros a un char, denominados 'car_in' y 'car_out'.      *
 * 'car_in' es la palabra de entrada que debe buscarse,       *
 * y 'car_out' es la palabra que debe devolverse a la         *
 * salida, si se encuentra 'car_in'                           *
 ***************************************************************
 */
struct mapa {
        char *car_in;
        char *car_out;
};

/***************************************************************
 * Define 'calles' como un array de estructuras 'mapa',       *
 * y lo inicializa a un conjunto de valores especificado      *
 * correspondientes a 'palabra-de-entrada' y 'palabra-de      *
 * salida'.                                                   *
 ***************************************************************
 */
struct mapa calles[] = {
        "ALCALDE",       "ALC",
        "ALDEREZ",       "ALFZ",
        "ALMIRANTE",     "ALMTE",
        "ARQUITECTO,"    "ARQTO"
        "DE",            (char *)0,
        "EL",            (char *)0,
        "NUESTRA",       "NRA",
        "PARQUE",        "PQE",
        "PRESIDENTE",    "PDTE",
        "PUERTO",        "PTO",
        "REPUBLICA",     "RPBCA",
        "SAN",           "S",
        "SANTA",         "STA",
        "SANTO",         "STO",
        "SEÑOR",         "SR",
        "SEÑORA",        "SRA",
        "TENIENTE",      "TTE",
        "VIRGEN",        "VG"
};

/***************************************************************
 * hi_calle es el numero de elementos de la estruct.calles    *
 * calculado como el tamaño en memoria de la estructura       *
 * dividido por el tamaño (sizeof) de un elemento de la       *
 * estructura. De este modo para añadir nuevos elementos      *
 * basta con incluirlos en su lugar en la estructura,         *
 * y el programa calculará automáticamente el número          *
 * de elementos                                               *
 ***************************************************************
 */
short hi_calle = sizeof(mapa) / sizeof(struct calles);
```



```
/***************************************************************
 * Programa principal. Notemos que 'main' va a manejar        *
 * argumentos de llamada, por eso escribimos                  *
 * main(argc, argv) y definimos argc y argv                   *
 ***************************************************************
 */
main(argc, argv)
int argc;                  /* define los argumentos de main */
char *argv[];
{
        unsigned i;     /* define variables locales a main */
        char *p;

/***************************************************************
 * Explora el argumento argv[1] mediante el puntero p,        *
 * considerando los caracteres ' ', '.' y ',' como            *
 * separadores de palabras                                    *
 ***************************************************************
 */
        for (i = 0, p = argv[1]; *p; p++) {
                switch (*p) {
                        case ' ':          /* separadores de */
                        case '.':          /* palabra        */
                        case ',':

/***************************************************************
 * Pone un caracter nulo '\0' como terminador de string       *
 * en la posición i del string palabra                        *
 ***************************************************************
 */
                                palabra[i] = '\0';

/***************************************************************
 * Concatena con 'calle' (resultado del programa) el valor    *
 * devuelto por la función 'tabla', que retorna un puntero    *
 * a un char, pudiendo ser tanto una palabra modificada       *
 * (si se ha encontrado en la estructura 'mapa'), como la     *
 * misma palabra a buscar, en caso de que no haya sido        *
 * encontrada en la búsqueda                                  *
 ***************************************************************
 */
                                strcat(calle, tabla(palabra));
                                i = 0;
                                break;

/***************************************************************
 * No separador: carga el caracter apuntado por p en la       *
 * posición i del array 'palabra', incrementando luego        *
 * la variable i, preparándola para la siguiente búsqueda     *
 ***************************************************************
 */
                        default:
                                palabra[i++] = *p;
                }
        }

/***************************************************************
 * Termina el string palabra, y realiza la última             *
 * búsqueda en la tabla de equivalencias                      *
 ***************************************************************
 */
        palabra[i] = '\0';
        strcat(calle, tabla(palabra));
```

```
/***********************************************************
 * Elimina (trima) los blancos múltiples que pueda         *
 * contener calle y muestra el nombre de entrada           *
 * y el obtenido como resultado del programa               *
 ***********************************************************
 */
        trimar(calle);
        printf("\nEntrada: ->%s<-", argv[1]);
        printf("\nSalida : ->%s<-\n", calle);
}



/***********************************************************
 * Función 'tabla', que retorna un puntero a un char       *
 ***********************************************************
 */
char * tabla(s)
char s[];           /* argumento de llamada a 'tabla' */
{

/***********************************************************
 * Define variables automáticas (sólo válidas en el        *
 * interior del cuerpo de la función 'tabla'):             *
 ***********************************************************
 */
        struct calles *p;
        int low, high, mid, cond;
        low = 0;
        high = hi_calle;

/***********************************************************
 * Búsqueda de 'palabra' en la tabla 'mapa'. Se emplea     *
 * una búsqueda binaria (binary search) por razones de     *
 * eficiencia, para disminuír el número de comparaciones   *
 * a realizar. La búsqueda binaria consiste en explorar    *
 * una tabla de nombres dispuestos en orden creciente,     *
 * de manera que en cada momento se compara sólo con el    *
 * elemento central de la tabla. Si el elemento a buscar   *
 * es > que el elemento central, entonces se seguirá bus-  *
 * cando en la mitad superior de la tabla, y,sinó en la    *
 * inferior.                                               *
 ***********************************************************
 */
        while (low <= high) {
                mid = (low + high) / 2;

/***********************************************************
 * Inicializa el puntero p para que apunte a la dirección  *
 * del elemento central de la tabla                        *
 ***********************************************************
                p = &mapa[mid];

/***********************************************************
 * Realiza la comparación, mediante la función de la       *
 * librería C estándar 'strcmp', que compara dos strings   *
 * entre sí, retornando un número > 0 si el primero es     *
 * mayor que el segundo, = 0 si son iguales, y < 0 si      *
 * el primero es menor que el segundo                      *
 ***********************************************************
 */
                if ((cond = strcmp(s, p->car_in)) < 0)
                        high = mid - 1;
```



```
                else if (cond > 0)
                        low = mid + 1;
                else

/* ¡Encontrado! */      return ((char *) p->car_out);

        }

/* No encontrado al final de la búsqueda */

        return ((char *) s);
}


/***********************************************************
 * Función 'trimar', que elimina los espacios en blanco    *
 * al principio y final de un string, así como los espa-   *
 * cios en blanco múltiples intercalados. Ante una entrada *
 * como:          "   ab   c  de  f  "                     *
 * retornaría:    "ab c de f"                              *
 ***********************************************************
 */
void trimar(s)
char *s;
{
        char *p;
        unsigned dbl = 0;

/***********************************************************
 * Elimina los espacios en blanco al principio del string  *
 ***********************************************************
 */
        for (p = str; *p == ' '; p++)
                ;

/***********************************************************
 * Explora el resto del string, poniendo el indicador      *
 * 'dbl' a 1 cuando encuentra un espacio en blanco, y      *
 * no copiando los caracteres a la salida. Cuando          *
 * encuentra un caracter distinto de blanco, pone          *
 * un espacio en blanco a la salida (si había blancos      *
 * múltiples) y copia los caracteres a la salida           *
 ***********************************************************
 */
        while (*p) {

/* Encuentra un espacio en blanco ' ' */

                if (*p == ' ') {
                        dbl = 1;
                        p++;
                        continue;
                }

/***********************************************************
 * Caracter distinto de espacio en blanco. Copia           *
 * blanco sobre la salida, pone a cero el indicador        *
 * 'dbl' y copia un caracter sobre la salida,              *
 * desplazando los punteros de entrada y salida            *
 ***********************************************************
 */
                if (dbl) {
                        *s++ = ' ';
```

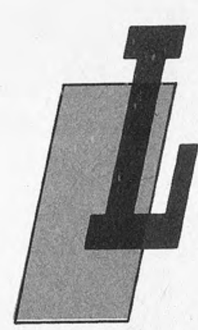
```
                dbl = 0;
            }
            *s++ = *p++;
        }

/****************************************************************
 * Pone un caracter nulo '\0' como terminador de string        *
 ****************************************************************
 */
        *s = '\0';
}
```



# APENDICE A

## *LA LIBRERIA C ESTANDAR*



La librería C estándar consiste en varios tipos de funciones para control de E/S, tratamiento de cadenas y caracteres, y funciones de tiempo y fecha. Todas las funciones son incluidas automáticamente por el compilador de lenguaje C, sin necesidad de utilizar opciones especiales de compilación.

La mayoría de las funciones requieren la utilización de algún fichero de #include, que debería incluirse al principio del (primer) fichero a compilar.

ABS - esta función, aplicada sobre una variable literal, nos devuelve su valor absoluto.

```
int abs(i)
int i;
```

ATOF - convierte una cadena de caracteres que contenga valores numéricos en su valor numérico expresado en doble precisión.

```
double atof(nptr)
char *nptr;
```

BSEARCH - búsqueda binaria.

```
char *bsearch((char *)key, (char *)base), nel,
    sizeof(*key), compar)
unsigned nel;
int (* compar) ();
```

CEIL - ante un valor numérico con decimales conserva la parte entera, despreciando los decimales.

```
#include <math.h>
double ceil(x)
double x;
```

CLOCK - devuelve el tiempo de CPU utilizado

```
long clock();
```

CONV - traslada caracteres.

```
#include <ctype.h>
int toupper(c)
int c;
```

o

```
#include <ctype.h>
int tolower(c);
int c;
```

o

```
#include <ctype.h>
int toascii(c)
int c;
```

CRYPT - genera encriptación por clave DES

```
char *crypt(key, salt)
char *key, *salt;

void settkey(key)
char *key;

void encrypt(block, edflag)
char *block;
int edflag;
```

CTERMID - asocia un nombre de fichero al terminal conectado.

```
#include <stdio.h>
char *ctermid(s);
char *s;
```

CTIME, LOCALTIME, GMTIME, ASCTIME, TIMEZONE - convierte la representación interna de la fecha y hora del sistema a un formato alfanumérico.

```
#include <time.h>
char *ctime(clock);
long *clock;
```

o

```
#include <time.h>
struct tm *localtime(clock);
long *clock;
```



o

```
#include <time.h>
struct tm *gmtime(clock);
long *clock;
```

o

```
#include <time.h>
char *asctime(tm);
struct tm *tm;
```

o

```
#include <time.h>
extern long timezone;
```

o

```
#include <time.h>
extern char *tzname;
```

o

```
#include <time.h>
void tzsc()
```

CTYPE - ordena los códigos ASCII correspondientes a valores enteros según una tabla.

```
#include <ctype.h>
int isalpha(c);
int c;
```

CURSES - funciones de manejo de pantalla con optimización de posicionamiento de cursor.

```
cc [flags] fichero -lcurses -ltermcap [librer
-lcurses rutinas de la librería curses
-ltermcap rutinas de la librería termcap
```

CUSERID - devuelve el nombre del usuario conectado al terminal.

```
#include <stdio.h>
char *cuserid(s)
char *s;
```

ECVT, FCVT, GCVT - realiza la conversión de un número en coma flotante a una cadena de caracteres.

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

o

```
char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

o

```
char *gcvt(value, ndigit, buf)
```

```
double value;
int ndigit;
char *buf;
```

FABS - devuelve el valor absoluto de un número.

```
#include <math.h>
double fabs(x)
double x;
```

FCLOSE, FFLUSH - cierra un canal de E/S o vacía su buffer.

```
#include <stdio.h>
int fclose(stream)
FILE *stream;
```

o

```
#include <stdio.h>
int fflush(stream);
FILE *stream;
```

FDOPEN - asocia un canal de E/S con un descriptor de fichero.

```
#include <stdio.h>
FILE *fdopen(fildes, type)
int fildes;
char *type;
```

FEOF - devuelve un valor distinto de cero cuando alcanza el final de fichero.

```
#include <stdio.h>
int feof(stream)
FILE *stream;
```

FERROR - devuelve un valor distinto de cero tras un error de lectura/escritura.

```
#include <stdio.h>
int ferror(stream);
FILE *stream;
```

FGETS - lee (n-1) caracteres de un canal de E/S.

```
#include <stdio.h>
char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;
```

FILENO - devuelve el número entero asociado al descriptor de fichero.

```
#include <stdio.h>
```



```
int fileno(stream);
FILE *stream;
```

FLOOR - ante un valor numérico con decimales devuelve el entero inmediatamente superior al representado sin decimales.

```
#include <math.h>
double floor(x)
double x;
```

FMOD - devuelve la función resto módulo.

```
#include <stdio.h>
double fmod(x, y)
double x, y;
```

FOPEN - abre un fichero y lo asocia con un canal de E/S.

```
#include <stdio.h>
FILE *fopen(filename, type)
char *filename, *type;
```

FPRINTF - escribe salida con formato sobre un canal de E/S.

```
#include <stdio.h>
int fprintf(stream, format [,arg] ...)
FILE *stream;
char *format;
```

FPUTC - escribe un carácter en un canal de E/S.

```
#include <stdio.h>
int fputc(c, stream);
FILE *stream;
```

FREAD, FWRITE - proporciona E/S binaria con buffer.

```
#include <stdio.h>
int fread(ptr, size, nitems, stream)
chat *ptr;
int size;, nitems;
FILE *stream;
```

o

```
#include <stdio.h>
int fwrite(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

FREOPEN - sustituye un canal de E/S abierto por un fichero.

```
#include <stdio.h>
FILE *freopen(filename, type, stream);
```

```
char *filename;
FILE *stream;
```

FSCANF - lee datos de un canal de E/S.

```
#include <stdio.h>
int fscanf(stream, format [,pointer] ...)
FILE *stream;
char *format;
```

FSEEK - posiciona el puntero para la próxima lectura/escritura en un canal de E/S.

```
#include <stdio.h>
int fseek(stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;
```

FTELL - devuelve la posición actual del puntero en un canal de E/S.

```
#include <stdio.h>
long ftell(stream);
FILE *stream;
```

GETC, GETCHAR, FGETC, GETW - toma un carácter o una palabra de un canal de E/S.

```
#include <stdio.h>
int getc(stream)
FILE *stream;
```

o

```
#include <stdio.h>
int getchar();
```

o

```
include <stdio.h>
int getw(stream)
FILE *stream;
```

GETENV - busca un nombre en el entorno de ejecución del usuario.

```
char *getenv(name)
char *name;
```

GETGRENT, GETGRGID, GETGRNAM, SETGRENT, ENDGRENT actúa sobre las protecciones de grupo de usuarios de un fichero.

```
#include <grp.h>
struct group *getgrent()
```



o

```
#include <grp.h>
struct group *getgrgid(gid)
int gid;
```

o

```
#include <grp.h>
struct group *getgrnam(name)
char *name;
```

o

```
#include <grp.h>
void setgrent()
```

o

```
#include <grp.h>
void endgrent()
```

GETLOGIN - devuelve el nombre del usuario conectado al terminal.

```
char *getlogin();
```

GETOPT - extrae las letras de opciones de un vector de argumentos.

```
int getopt(argc, argv, optstring)
int argc;
char **argv;
char *opstring;
extern char *optarg;
extern int optind;
```

GETPASS - lee la clave de acceso (password) de un usuario.

```
int getpw(uid, buf)
int uid;
char *buf;
```

GETPWENT, GETPWUID, GETPWNAM, SETPWENT, ENDPWENT actúa sobre las entradas del fichero de usuarios (passwd).

```
#include <pwd.h>
struct passwd *getpwent()
```

o

```
#include <pwd.h>
struct passwd *getpwuid(uid)
int uid;
```

o

```
#include <pwd.h>
struct passwd *getpwnam(name)
char *name;
```

o

```
#include <pwd.h>
void setpwent()
```

o

```
#include <pwd.h>
void endpwent()
```

GETS - lee un literal de un canal de E/S.

```
#include <stdio.h>
char *gets(s);
char *s;
```

LOGNAME - devuelve el nombre de conexión (login) de un usuario.

```
char *logname()
```

LONGJMP - recupera el stack del entorno de usuario salvado previamente.

```
#include <setjmp.h>
void longjmp(env, val)
jmp_buf env;
int val;
```

MALLOC, FREE, REALLOC, CALLOC - asignadores dinámicos de memoria central.

```
char *malloc(size)
unsigned size;
```

o

```
void free(ptr)
char *ptr;
```

o

```
char *realloc(ptr, size)
char *ptr;
unsigned size;
```

o

```
char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

MKTEMP - crea un nombre de fichero garantizando que sea único.

```
char *mktemp(template)
char *template;
```

PERROR, ERRNO, SYS_ERRLIST, SYS_NERROR - trata los mensajes de error del sistema.

```
void perror(s)
char *s;
```



o

```
extern int errno;
```

o

```
extern char *sys_errlist[];
```

o

```
extern int sys_nerr;
```

POPEN, PCLOSE - inicia o finaliza la E/S de un proceso a través de un pipe (tubería).

```
#include <stdio.h>
FILE *popen(command, type)
char *command, *type;
```

o

```
#include <stdio.h>
int pclose(stream)
FILE *stream;
```

PRINTF - imprime salida con formato.

```
#include <stdio.h>
int printf(format [,arg] ...)
char *format;
```

PUTC - pone un carácter en un canal de E/S.

```
#include <stdio.h>
int putc(c, stream)
char c;
FILE *stream;
```

PUTCHAR - pone un carácter en el fichero de salida estándar (stdout), por defecto la pantalla.

```
#include <stdio.h>
int putchar(c)
char c;
```

PUTS, FPUTS - pone un literal en un canal de E/S.

```
#include <stdio.h>
int puts(s)
char *s;
```

o

```
#include <stdio.h>
int fputs(s, stream)
char *s;
FILE *stream;
```

PUTW - pone una palabra en un canal de E/S.

```
#include <stdio.h>
```

```
int putw(w, stream)
int w;
FILE *stream;
```

RAND, SRAND - genera números aleatorios.

```
int rand()
```

o

```
void srand(seed)
unsigned seed;
```

REWIND - posiciona al principio el puntero de acceso a un fichero.

```
#include <stdio.h>
void rewind(stream)
FILE *stream;
```

SCANF - lee del canal de entrada estándar (stdin).

```
#include <stdio.h>
int scanf(format [,pointer] ...)
char *format;
```

SETBUF - asigna un buffer a un canal de E/S.

```
#include <stdio.h>
void setbuf(stream, buf)
FILE *stream;
char *buf;
```

SETJMP - guarda el stack de ejecución de un proceso.

```
#include <setjmp.h>
int setjmp(env)
jump_buf env;
```

SLEEP - suspende la ejecución de un proceso durante un intervalo de tiempo especificado.

```
unsigned sleep(seconds)
unsigned seconds;
```

SPRINTF - escribe en un literal con salida formateada.

```
#include <stdio.h>
int sprintf(s, format [,arg] ...)
char *s, *format;
```

SSCANF - lee un literal con formato.

```
#include <stdio.h>
int sscanf(s, format [,pointer] ...)
char *s, *format;
```



STDIO - E/S estándar bufferizada.

```
#include <stdio.h>
FILE *stdin, *stdout, *stderr;
```

STRING - realiza operaciones con literales (strings).

STRCAT - añade una copia del literal s2 al final del literal s1

```
#include <stdio.h>
char *strcat(s1, s2)
char *s1, *s2;
```

STRNCAT - añade una copia de "n" caracteres del literal s2 al final del literal s1

```
#include <stdio.h>
char *strncat(s1, s2, n)
char *s1, *s2;
int n;
```

STRCMP - compara los literales s1 y s2

```
#include <stdio.h>
int strcmp(s1, s2)
char *s1, *s2;
```

STRNCMP - compara al menos "n" caracteres de los literales s2 y s1

```
#include <string.h>
int strncmp(s1, s2, n)
char *s1, *s2;
int n;
```

STRCPY - copia el literal s2 sobre el literal s1

```
#include <string.h>
char *strcpy(s1, s2)
char *s1, *s2;
```

STRNCPY - copia exactamente "n" caracteres del literal s2 al literal s1

```
#include <string.h>
char *strncpy(s1, s2, n)
char *s1, *s2;
int n;
```

STRLEN - devuelve el número de caracteres distintos de nulo

```
#include <string.h>
int strlen(s)
char *s;
```

STRCHR - devuelve un puntero a la primera ocurrencia del carácter "c" en el literal "s"

```
#include <string.h>
char *strchr(s, c)
char *s, c;
```

STRRCHR - devuelve un puntero a la última ocurrencia del carácter "c" en el literal "s"

```
#include <string.h>
char *strrchr(s, c)
char *s, c;
```

STRPBRK - devuelve un puntero al primer carácter del literal s1 encontrado en el literal s2

```
#include <stdio.h>
char *strpbrk(s1, s2)
char *s1, *s2;
```

STRSPN - devuelve la longitud del literal s1 que coincide con el literal s2

```
#include <string.h>
int strspn(s1, s2)
char *s1, *s2;
```

STRCSPN - devuelve la longitud del literal s1 que no coincide con el literal s2

```
#include <string.h>
int strcspn(s1, s2)
char *s1, *s2;
```

STRTOK - devuelve un puntero a la primera ocurrencia del literal s1 en el literal s2

```
#include <string.h>
char *strtok(s1, s2)
char *s1, *s2;
```

STRTOL, ATOL, ATOI - convierten un literal en un entero

```
long strtol(str, ptr, base)
char *str;
char **ptr;;
int base;
```

o

```
long atol(str)
char *str;
```

o

```
int atoi(str)
char *str;
```



SWAB - intercambia bytes.

```
void swab(from, to, nbytes)
char *from, *to;
int nbytes;
```

SYSTEM - proporciona un comando de la shell.

```
#include <stdio.h>
int system(string)
char *string;
```

TERMCAP - proporciona subrutinas para manejo del terminal, especificando las características de los atributos de vídeo propios de cada terminal.

TEGENT - pone el nombre del terminal en un buffer

```
tgetent(bp, name)
char *bp, *name;
```

TGETNUM - devuelve el valor numérico del atributo "id" del terminal

```
#tgetnum(id)
char *id;
```

TGETSTR - toma el valor literal del atributo "id" del terminal

```
char * tgetstr(id, area)
char *id, **area;
```

TGOTO - devuelve un literal para efectuar el posicionamiento del cursor del terminal

```
char *tgoto(cm, destcol, destline)
char *cm;
int destcol, destline;
```

TMPFILE - crea un fichero temporal.

```
#include <stdio.h>
FILE *tmpfile();
```

TMPNAME, TEMPNAM - asigna un nombre a un fichero temporal.

```
#include <stdio.h>
char *tmpnam(s)
char *s;
```

o

```
#include <stdio.h>
char *tempnam(dir, pfx)
char *dir, *pfx;
```

TTYNAME, ISATTY - proporciona el nombre del terminal.

```
char *ttyname(fildes)
int fildes;
```

o

```
int isatty(files)
int fildes;
```

UNGETC - devuelve un carácter a un canal de E/S.

```
#include <stdio.h>
int ungetc(c, stream)
char c;
FILE *stream;
```



# BIBLIOGRAFIA


Programación en C: introducción y conceptos avanzados.
Waite, Prata y Martin. *Anaya Multimedia.* Madrid, 1985.

C. Guía de Programación.
Jack J. Purdum. *Díaz de Santos.* Madrid, 1985.

El lenguaje de Programación C.
Brian W. Kernigham y Dennis M. Ritchie. *Prentice-Hall.* New Jersey, 1978.

The C Programmers Handbook.
AT&T y Bell. 1985.

The C puzzle Book.
Alan R. Feuer. *Prentice-Hall.* Englewood Cliffs, New Jersey, 1982.

Advanced C: Food for the educate palate.
N. Gehani, 1985.

C: a reference manual.
Harbison y Steele, 1984.

C Programming Standards and Guidelines: Version U (UNIX and offspring).
Thomas Plum. *Plum Hall.* Cardiff, New Jersey, 1981.

Thomas Plum. *Plum Gall.* Cardiff, New Jersey, 1981. "C Programming Standars and Guidelines: Versión W (Whitesmiths)".

Learning to program in C.
Thomas Plum. *Plum Hall,* 1983.

C Programmers Library.
Jack J. Purdum, Timothy C. Leslie y Alan L. Stegemoller. *Que Corporation.* Indianápolis, Indiana, 1984.

The C Programming Language.
Dennis M. Ritchie, S. C. Johnson, M. E. Lesk y Brian W. Kernigham. *The Bell System Technical Journal* (BSTJ). Vol. 57, No. 6, julio-agosto 1978.

Programming in C for the microcomputer user.
R. J. Traister. 1984.

The C Programming Tutor.
Leon A. Wortman y Thomas O. Sidebottom. *Prentice-Hall.* Englewood Cliffs, New Jersey, 1984.

C Notes.
Carrol Zahn. *Yourdon Press.* New York. N.Y., 1978.



# BIBLIOTECA BASICA INFORMATICA

INDICE GENERAL

10 **Sistemas operativos y software de base**
Qué son, para qué sirven. Unos desconocidos muy importantes.
11 **Sistema operativo CP/M**
Uno de los sistemas operativos para microprocesadores de 8 bits de mayor difusión en el mercado.
12 **MS-DOS: el estándar de IBM**
Sistema operativo para el microprocesador de 16 bits 8088, adoptado por el IBM-PC.
13 **Paquetes de aplicaciones. Software "pret a porter"**
Características y peculiaridades de los más importantes paquetes de aplicaciones.
14 **VisiCalc: una buena hoja de cálculo**
Interioridades y manejo de una de las hojas de cálculo más usadas.
15 **Dibujar con el ordenador**
Profundizando en una de las facetas útiles y divertidas que nos ofrecen los ordenadores.
16 **Tratamiento de textos... para escribir con el ordenador**
Cómo convertir su ordenador en una máquina de escribir con memoria y todo tipo de posibilidades.
17 **Diseño de juegos**
Particularidades características de esta aplicación de los ordenadores.
18 **LOGO: la tortuga inteligente**
Un lenguaje conocido por su «cursor gráfico», la tortuga, y sus aplicaciones pedagógicas al alcance de su mano.
19 **Paquetes integrados: Lotus 1-2-3 y Simphony**
Estudio de dos de los paquetes integrados (Hoja de cálculo+base de datos+...) más conocidos.
20 **dBASE II y dBASE III**
Cómo aprovechar las dos versiones más recientes de esta importante base de datos.
21 **Bancos de datos (I)**
Peculiaridades de una de las aplicaciones de los ordenadores más interesantes y que más dinero mueven.
22 **Bancos de datos (II)**
Profundizando en sus características.
23 **FORTH: anatomía de un lenguaje inteligente**
Principales características de un lenguaje moderno, flexible y de amplio uso, en la robótica.
24 **BASIC y tratamiento de imágenes**
Todo lo que en **¡Y llegó el BASIC!** no se pudo ver sobre las imágenes y gráficos en el BASIC.
25 **Los ordenadores uno a uno**
Un amplio y completo estudio comparativo.
26 **Cálculo numérico en BASIC**
Una aplicación especializada a su disposición.
27 **Multiplan**
Cómo hacer uso de este moderno paquete de aplicaciones.
28 **FORTRAN y COBOL**
Dos lenguajes muy especializados y distintos.
29 **Softest.** Los programas a examen
30 **Cómo realizar nuestro propio banco de datos**
Conocimientos necesarios para poder fabricar un banco de datos a nuestro gusto y medida.

**NOTA: Ingelek, S. A. se reserva el derecho de modificar, sin previo aviso, el orden, título o contenido de cualquier volumen de la colección.**

# NOTAS

*Cuando oímos hablar de lenguajes de programación los que se nos vienen a la memoria son el BASIC, el FORTRAN, el PASCAL u otros. Sin embargo, el C se ha ganado en poco tiempo el derecho a entrar en esta lista de "habituales". Son pocas las personas que se mueven alrededor de los ordenadores personales y supermicros que no han oído hablar de este lenguaje y de sus posibilidades.*

*Una de las particularidades que más le ha elevado a la "cresta de la ola" es su íntima ligazón con el sistema operativo Unix, que se está convirtiendo en un estándar para los supermicros de 16, 32 y, en el futuro, 64 bits.*

*Este volumen de la B.B.I. pretende ser una primera introducción al C. dando una primera idea de su desarrollo, características y evolución futura, de forma que queden sentadas las bases que faciliten el acceso a libros más especializados*

*395 pts*

*(incluido IVA*